Sentence Analysis Techniques in Speech Translation(Dissertation_全文)

AUTHOR(S):

Tomita, Masaru

(2)

Sentence Analysis Techniques in Speech Translation

Masaru Tomita

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Speech translation (voice input and voice output) is much more than connecting a speech recognizer to a machine translation system with a speech synthesizer. Suppose you have these three components and each individual component works well. If you simply connect the three to perform speech translation, you probably end up with very poor speech translation.

There are two main problems. First, no speech recognizer is perfect, and spoken input sentences are often recognized with errors. A certain word in the input sentence may be misrecognized as a different word, or may be completely ignored, resulting an ungrammatical sentence.

Second, people rarely speak grammatical sentences to begin with. Unlike written sentences, spoken sentences often include nonsense words and phrases, unnecessary repetition, meaningless pauses, cough and other noises, as seen in the following example sentence: "Is this well is this ah the conference desk I mean conference office?"

Most conventional machine translation systems, which expect their input sentences to be grammatical, can do very little with those ungrammatical sentences; they probably produce very funny translation, or they simply crash.

This document describes several techniques to solve some of these problems.

Chapter 2 describes the Generalized LR parsing algorithm for context-free grammars. This is the algorithm which is the basis of most of the techniques presented later the documeut. Generalized LR parsing uses the precompiled LR parsing table, and unlike LR parsing, it can handle arbitrary context-free grammars while most of the LR efficiency is preserved with the

device called *Graph Structured Stack*.

Chapter 3 then describes how to handle augmented context-free grammars (context-free grammars which have additional conditions and actions attached to each rule) using Generalized LR parsing. A practical grammar formalism, which resembles LFG, is introduced, and how the grammar can be precompiled is presented.

Analysis of spokeu sentences with possible recognition errors can be highly ambiguous, and some kind of mechanism is definitely needed to tell which is the most likely parse out of multiple ambiguous parses. One such mechanism is using probabilistic grammars, where each grammar rule has a probability value indicating how likely the rule is used. Chapter 4 has detailed discussions of how we can handle probabilistic grammars efficiently using Generalized LR parsing; specifically, how one can precompile such probabilistic information into the LR parsing table.

Some speech recognition systems produce not just a sequence of words, but a *lattice* of word candidates called *word lattice*. A word lattice is an efficient representation of a large number of sentence candidates. In parsing word lattices, the search space is much larger than in parsing word strings (sentences), and an efficient algorithm is required. Chapter 5 presents an efficient word lattice parsing algorithm based on Generalized LR parsing. The algorithm can parse input in any order; thus acoustically more reliable words and semantically more significant words can be processed first.

Chapter 6 describes an extension of Generalized LR parsing that is capable of skipping unrecognizable parts of the input sentence. Spontaneously spoken sentences are very ungrammatical, and it is next to impossible to write a grammar which covers them. Therefore, rather than trying to write a huge grammar to cover spontaneous utterances, we want to write a concise grammar and have the parser extract parts of the utterance that are meaningful and ignore the rest. Such a parsing algorithm, GLR*, is presented in the chapter, and some heuristics to make it more efficient are discussed.

In chapter 7, we describe three different speech translation projects, which the author has been involved. *SpeechTrans* is a Japanese to English speech translation system, in the domain of doctor patient conversation with the vocabulary size of 100. *Sphinx-LR* is an English to Japanese speech translation system in the domain of resource management with the vocabulary size of 1000. It uses a speech recognition system named *Sphinx* which was developed at Carnegie Mellon University. Then, *JANUS* is an English to German/Japanese speech translation system in the domain of conference registration with the vocabulary size of 500. It uses a speech recognition

system based on neural networks which was also developed at Carnegie Mellon University. All of the three systems are *speaker independent* and for *continuous speech*.

Finally in chapter 8, some concluding remarks are made.

In the appendices, there are user's manuals of two computer software packages developed by the author: *The GLR Parser/Compiler* version 8-4 and *GENKIT and TRANSKIT* version 3-2. Those software packages are available to the public, and by now, dozens of projects, world wide, have been using them as a tool for implementing natural language systems.

Acknowledgements

I would like to thank Professor Makoto Nagao, for the opportunity he has given me to submit this document for a degree of Doctor of Engineering, and more importantly, for his continuous support, technical and personal, for many years during my career in the area of speech translation.

Substantial parts of this thesis are based on the author's previously published papers, as indicated in the following list.

- Chapter 2: [Tomita, 1991, Tomita and Ng, 1991, Tomita, 1987, Tomita, 1988b],
- Chapter 3: [Tomita, 1987, Tomita, 1990b],
- Chapter 4: [Ng and Tomita, 1991],
- Chapter 5: [Lavie and Tomita, 1993a, Tomita, 1986, Lavie and Tomita, 1993b],
- Chapter 6: [Lavie and Tomita, 1993c],
- Chapter 7: [Tomita *et al.*, 1989, Nirenburg *et al.*, 1991, Tomita *et al.*, 1990a, Saito and Tomita, 1988b, Woszczyna *et al.*, 1993, Tomita, 1988e, Tomabechi *et al.*, 1989, Tomita *et al.*, 1990b],
- Appendix A: [Tomita *et al.*, 1988b, Tomita, 1990b],
- Appendix B: [Tomita and Nyberg, 1988].

Other published papers that are closely related to this thesis are [Tomita, 1990a, Tomita, 1991, Nirenburg *et al.*, 1991, Tomabechi and Tomita, 1989, Tomita and Carbonell, 1986, Tomita *et al.*, 1987, Tomita and Carbonell,

1987b, Tomita, 1988a, Tomita *et al.*, 1988a, Kitano *et al.*, 1989, Tomita, 1988c, Tomita, 1988d, Carbonell and Tomita, 1985, Tomita and Carbonell, 1988]. A part of Chapter 7 is based on [?].

I wish to acknowledge the authors of these papers, parts of which, directly or indirectly, appear in this document. They are O. Barkai, Jaime Carbonell, Noah Coccaro, A. Eisele, Ken Goodman, Marion Kee, T. Kawabata, Kenji Kita, Hiroaki Kitano, Alon Lavie, Arthur McNair, Teruko Mitamura, Hiroyuki Musha, See-Kiong Ng, Sergei Nirenburg, Eric Nyberg, I. Rogina, Carolyn Rose, Hiroaki Saito, T. Sioboda, Hideto Tomabechi, Naomi Waibel, Alex Waibel, Wayne Ward, and Monica Woszczyna.

There are many other people who gave me technical and/or personal supports. They include Hideo Aiso, Shuji Doshita, Hitoshi Iida, Akira Kurematsu, Lori Levin, Joan Maddamma, Shuji Morii, Seiichi Nakagawa, Shinya Nakagawa, Masakazu Nakanishi, Tadashi Nakayama, Allen Newell, Toyoaki Nishida, Raj Reddy, Kiyohiro Shikano, Herbert Simon, Hozumi Tanaka, Junya Tsutsumi, and Jun-ichi Tsujii.

Finally, I would like to thank my wife, Yuko, and my children, Ellie and Ken for their continuous emotional support and encouragement.

Chapter 2

Generalized LR Parsing

2.1 Introduction

LR parsing is a widely used method of syntax analysis for a variety of reasons [1].

First and foremost, an LR parser is a deterministic parser which is highly efficient: it scans the input string in one pass and is able to detect errors at an early stage. The availability of parser generators and compiler compilers based on LR parsing technology [Aho and Ullman, 1977, Johnson, 1975] further accentuated its popularity, since such tools are essential in practical systems where grammars are often too large for a parser to be constructed by hand.

A major drawback of standard LR parsing is that it can only handle a subclass of context-free grammars called *LR grammars*. Although there exist parsing techniques capable of handling arbitrary context-free grammars, for instance, the Earley's algorithm [Earley, 1970] and the Cocke-Younger-Kasami algorithm [Kasami, 1965], these algorithms often do not perform as well as LR parsing.

One of the strong points of standard LR parsing is that it is totally deterministic, which gives rise to its efficiency in execution. In order to achieve determinism, the LR parser sacrifices its generality by imposing a stringent condition on the class of grammars over which the technique works. It requires that an LR parsing table with no action conflicts be producible for

[1] Major parts of this chapter are based on previously published papers [Tomita, 1991, Tomita and Ng, 1991, Tomita, 1987, Tomita, 1988b]. I would like to acknowledge the coauthor of the papers, See-Kiong Ng, whose contribution is included in this chapter.

the grammar. It is conceivable that in some practical applications, such grammars are hard to come by. For example, in natural language processing, there are grammatical features such as prepositional phrase attachment which are inherently ambiguous, causing any context-free grammar which models such feature to be non-LR. If there is a general and efficient method for dealing with action conflicts in the parsing table, then there lies an efficient parsing algorithm for general context-free grammar. Generalized LR (GLR) parsing, which was introduced by the author [Tomita, 1985, Tomita, 1987], is one such technology. By using a *graph-structured stack* to simulate nondeterminism, GLR parsers are able to handle general context-free grammars while retaining much of the efficiency of standard LR parsing (especially when the grammar is close to being LR). Other deterministic techniques for non-deterministic context-free parsing have been developed independently by Lang [Lang, 1974] and van der Steen [Van der Steen, 1987].

In the following sections, we will first describe the basic notion of the graph-structured stack as a general mechanism. Then, we describe a compacted way of representing the possible parse trees for an ambiguous sentence, known as *packed shared parse forest*. Next, to give the reader a fair idea of how a GLR parser actually works, a detailed example trace of a GLR parser on an ambiguous sentence is given. Finally, a specification of the GLR algorithm is presented. In all the discussions, we shall assume that the reader is familiar with the standard LR parsing technique. Extensive descriptions of standard LR parsing can be found in [Aho and Ullman, 1972, Aho and Ullman, 1977].

2.2 Graph-structured Stack

The *graph-structured stack* is a general device for efficient handling of nondeterminism in parsing systems employing a stack. In this section, we shall describe three key notions of the graph-structured stack, namely splitting, combining and local ambiguity packing.

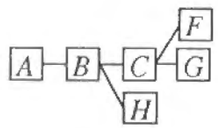
2.2.1 Splitting

When a stack can be reduced (or popped) in more than one way, the top of the stack is made to split to accommodate the various possibilities. Consider the following example. The current stack configuration is displayed below: the stack is represented left to right, that is, the leftmost element A is the bottom of the stack, and the rightmost element E the top of the stack.

Now suppose that the stack is to be reduced with each of the following three productions in parallel:

$$F \rightarrow D\ E$$
$$G \rightarrow D\ E$$
$$H \rightarrow C\ D\ E$$

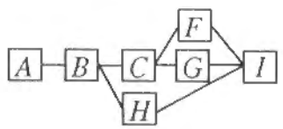
After the nondeterministic reduce actions, the stack has the following form:



Since the stack has a graph structure, it can have more than one stack top. A stack top in a graph-structured stack, in our left-to-right representation, is a stack node with no nodes attached to its right. In the above example, F, G and H are the stack tops.

2.2.2 Combining

When an element needs to be shifted (pushed) onto more than one stack top, it is done only once by combining the tops of the stack. As a continuation to the previous example, if I is to be shifted to F, G and H, then the resulting stack will look like:

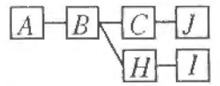


2.2.3 Local Ambiguity Packing

If two or more branches of the stack turn out to be identical, then they represent local ambiguity. That is, the identical state of the stack has been obtained in two or more different ways. These branches are merged and treated as a single branch. Continuing from the above example, suppose we are now to reduce the stack by each of the following productions nondeterministically:

$$J \rightarrow F\ I$$
$$J \rightarrow G\ I$$

Instead of having two identical J nodes sprouting from C, the resulting stack looks as follows:



The branch A–B–C–J has been obtained in two ways, but they are packed together so that only a single branch remains on the stack.

2.3 Packed Shared Parse Forest

For a highly ambiguous grammar, there might be a humongous number of possible parse trees for an input sentence. Instead of storing each of the parse trees separately, we can exploit the efficient operations of the graph-structured stack to build a *packed shared parse forest* which represents the numerous possible parse trees in a space-efficient manner. To avoid confusion in the discussion, we shall use the term *vertex* for parse forests, whereas *nodes* shall be reserved for elements on a graph-structured stack.

First of all, if two or more trees have a common subtree, the subtree should be represented only once in the forest. We call this *subtree sharing* and a parse forest with such property is called a *shared* parse forest.

We can further minimize the representation of the parse forest by *local ambiguity packing*, which works in the following way. The top vertices of subtrees that represent local ambiguity are merged and treated as if there were only one vertex. We call such a vertex a *packed vertex*. A parse forest with both subtree sharing and local ambiguity packing is called a *packed shared forest*. Figure 2.17 shows a packed shared parse forest for the sentence *"I saw Jane and Jack hit the man with a telescope"*, in which the packed vertices are boxed. From this example, one might observe that a packed shared parse forest is effectively a *tree* in which some of thenonterminal vertices (the packed vertices) have several sets of children. Each of the children sets corresponds to a possible derivation of the nonterminal represented by that vertex based on the same input. It is easy to see how the numerous parse trees can be enumerated from the packed shared parse forest.

It turns out that GLR parsing provides a natural way to construct a packed shared parse forest during parse time. To implement subtree sharing, we push pointers to a vertex of the shared parse forest together with the stack node. That is, when the parser *shifts* a terminal in the input, it creates a leaf vertex (if it has not already been created) labeled with that terminal and pushes the pointer to this vertex together with the stack node onto the stack.

When the parser *reduces* the stack, it pops pointers from the stack, creates a new vertex whose children are the vertices pointed to by these pointers, and pushes the pointer to this new vertex together with the stack node onto the stack. Packed vertices, on the other hand, are created naturally by the process of *local ambiguity packing* of the graph-structured stack. Recall that local ambiguity packing enusres that a single stack node is employed to represent multiple derivations for the nonterminal represented by the node. With such a stack node located by the process, we can correspondingly pack the parse forest vertex in the stack node by appending the new set of children vertices for the current derivation to the vertex instead of creating a brand new vertex for the reduction. This process of constructing a packed shared forest with GLR parsing would become clear to the reader with the example in the next section.

2.4 An Example

Figure 2.1 shows a non-LR context-free grammar GRA which contains a fair amount of ambiguity.

(1)	$S \rightarrow NP\ VP$
(2)	$S \rightarrow S\ PP$
(3)	$S \rightarrow S\ and\ S$
(4)	$NP \rightarrow n$
(5)	$NP \rightarrow det\ n$
(6)	$NP \rightarrow NP\ PP$
(7)	$NP \rightarrow NP\ and\ NP$
(8)	$VP \rightarrow v\ NP$
(9)	$VP \rightarrow v\ S$
(10)	$PP \rightarrow p\ NP$

Figure 2.1: GRA: A non-LR Grammar

GRA is a toy grammar for modeling conjunctive phrases with prepositional attachments in English. Naturally, prepositional attachments and conjunctive grouping are the two main sources of ambiguities here. The former can be exemplified by the sentence "I saw a man with a telescope", for which there are two different interpretations due to ambiguous prepositional

attachment:

1. [I saw [a man with a telescope]]
2. [I saw [a man] with a telescope]

Ambiguity due to conjunctive grouping can be illustrated by the sentence "I know Jane and Jack knew it":

1. I know [Jane and Jack knew it].
2. [I know Jane] and [Jack knew it].

The interaction between these two sources of ambiguities worsens the situation, such as in the sentence "I saw Jane and Jack hit the man with a telescope". Sometimes, these ambiguities may be resolved by using punctuations in the text (for example, by inserting a comma at the boundaries of conjunctive groupings), intonation in speech, or semantic and contextual information. The problem is that such additional information may not be available at parse time. The parser is often obliged to generate all syntactically consistent interpretations until further information can be used for disambiguation.

Figure 2.2 shows an SLR(1) parsing table for GRA as constructed using the simple-LR table construction method described in [Aho and Ullman, 1972, Aho and Ullman, 1977]. It does not have to be SLR(1); it can be LR(0), LR(1), LALR(1), LR(2), SLR(2), or any other variation of LR table construction method [Aho and Ullman, 1977]. The parser described in this chapter can work with any kind of shift-reduce LR parsing table. [2]

The parsing table consists of two parts: an ACTION table and a GOTO table. The ACTION table is indexed by a state symbol s (row) and a terminal symbol x (column), including the end marker \$. The entry ACTION$[s, x]$ can be one of the following: *sh*, *re n*, *acc* or blank. *sh* denotes a shift action, *re n* means a reduction by the n-th production, *acc* denotes the accept action and a blank indicates a parsing error. The GOTO table is indexed by a state symbol s (row) and a grammar symbol X (column). The entry GOTO$[s, X]$ defines the next state the parser should go to. During parse time, the parser consults the ACTION table for parsing actions to execute based on its current configuration, executes the actions accordingly, and then refers to the GOTO table for its next state.

[2] However, more than one lookahead (LR(k) with k>1) is considered not practical, as its parsing table becomes very large with little gain of run time efficiency.

State	ACTION						GOTO								
	det	*n*	*v*	*p*	*and*	*$*	*det*	*n*	*v*	*p*	*and*	*S*	*NP*	*VP*	*PP*
0	*sh*	*sh*					*2*	*1*				*4*	*3*		
1			*re4*	*re4*	*re4*	*re4*									
2		*sh*						*5*							
3			*sh*	*sh*	*sh*				*6*	*7*	*8*			*9*	*10*
4				*sh*	*sh*	*acc*				*7*	*11*				*12*
5			*re5*	*re5*	*re5*	*re5*									
6	*sh*	*sh*					*2*	*1*				*14*	*13*		
7	*sh*	*sh*					*2*	*1*					*15*		
8	*sh*	*sh*					*2*	*1*					*16*		
9				*re1*	*re1*	*re1*									
10			*re6*	*re6*	*re6*	*re6*									
11	*sh*	*sh*					*2*	*1*				*17*	*3*		
12				*re2*	*re2*	*re2*									
13			*sh*	*sh, re8*	*sh, re8*	*re8*			*6*	*7*	*8*			*9*	*10*
14				*sh, re9*	*sh, re9*	*re9*				*7*	*11*				*12*
15			*re10*	*sh, re10*	*sh, re10*	*re 10*				*7*	*8*				*10*
16			*re7*	*sh, re7*	*sh, re7*	*re 7*				*7*	*8*				*10*
17				*sh, re3*	*sh, re3*	*re3*				*7*	*11*				*12*

Figure 2.2: SLR(1) Parsing Tahle for GRA

In the following, we give a trace of the GLR algorithm on the input sentence "I saw Jane and Jack hit the man with a telescope". In each step of the trace, we show the following:

- **The graph-structured stack:** Each stack node is represented either as a square or a circle with a state number in it and the corresponding parse forest vertex above it. An active top node is depicted as a circle, with the pending parsing actions placed next to it. Although we do not actually delete the stack nodes duriug reduction, we do not display the irrelevant nodes in our trace diagrams for the sake of clarity.

- **The packed shared parse forest:** Each vertex in the parse forest is named as X_n, where X is a grammar symbol represented by that vertex, and n is a unique subscript to distinguish between different vertices representing X in the parse forest. An ordinary vertex is represented as a dot, whereas a packed vertex is represented as a highlighted box encompassing the dots (vertices) that represent the various possible parses for the locally ambiguous symbol. On the parse stack, a star is placed momentarily beside the respective parse vertex to indicate where and when local ambiguity packing has occurred.

- **Next input word:** We indicate the next word of the input sentence at the top of each trace diagram. It is shown as a pair *"w":x* , where w is the actual word in the input sentence, and x a terminal symbol in the grammar which represents the lexical category of w. We assume that every word can be categorized unambiguously, although the parser could easily handle lexical ambiguities by treating the various possibilities as action conflicts [Tomita, 1985].

- **Next ACTIONs and GOTO states:** The parsing actions, which are placed heside the respective active top nodes, are specified as pairs $[a, s]$, where a is the ACTION to be performed on the node, and s is the corresponding GOTO state after executing a. Let us call a node that immediately precedes a reduction path on the stack a *base* node. For a reduction which splits a merged node, there are more than one base node. In this case, s is a column which represents each of the base nodes' goto states in the top-down order in which the respective base nodes are heing depicted in the graph.

We are now ready to begin the trace of the GLR parser ou the sentence:

"I saw Jane and Jack hit the man with a telescope".

Initially, the stack contains only one node with state 0, and the parse forest is null (represented as $\perp$). The next word is "I", which is categorized as a noun n. Since ACTION$[0, n] = sh$ and GOTO$[0, n] = 1$, we place the pair $[sh, 1]$ next to the node, which is denoted by a circle since it is currently an active stack top (see Figure 2.3).

Next word = "I" : n

Figure 2.3: Trace of the parser

In executing the shift action, the parser creates a parse forest vertex n_1 for the word "I", and pushes a stack node of state 1 and vertex n_1 onto the stack. The next word is "saw", which is a verb v. Since ACTION$[1, v] = re4$ and GOTO$[0, NP] = 3$ (as production 4 is $NP \rightarrow n$ and the base node for this reduction is the start node, which has state 0), this new node is active with the pair $[re4, 3]$, as shown in the first row of Figure 2.4. The second row shows the resulting configuration of the parser after this reduce action is executed: a new parse forest vertex NP_1 with child n_1 is created, the stack node along the reduction path is popped, and a new node with state 3 and vertex NP_1 is pushed onto the stack. The action-goto pair for this new active stack top is $[sh, 6]$.

After the shift action for the word "saw" is executed, the resulting configuration calls for another shift action for the input "Jane", as depicted in Figure 2.5.

After the word "Jane" is shifted onto the stack, the next word is the conjunction "and". The first action to execute is "*re 4*" (see the first row of Figure 2.6). This reduction results in a new stack node of state 13 and parse forest vertex NP_2, as shown in the second row of the figure. At this node, the parser encounters, for the first time, a parsing action conflict, since ACTION$[13, and]$ can be "*sh*" or "*re 8*". Since the GLR algorithm requires that all the current reduce actions be processed before the shifts, the parser leaves this top node active with the shift action for now. The "*re 8*" action is executed, sprouting a new branch on the stack (see the third row

Next word = "saw" : v

Figure 2.4: Trace of the parser (cont'd)

Next word = "Jane" : n

Figure 2.5: Trace of the parser (cont'd)

of Figure 2.6). A further reduction of the new branch occurs, after which both branches of the stack are left with pending shift actions, as in the last row of the figure.

Next word = "and" : and

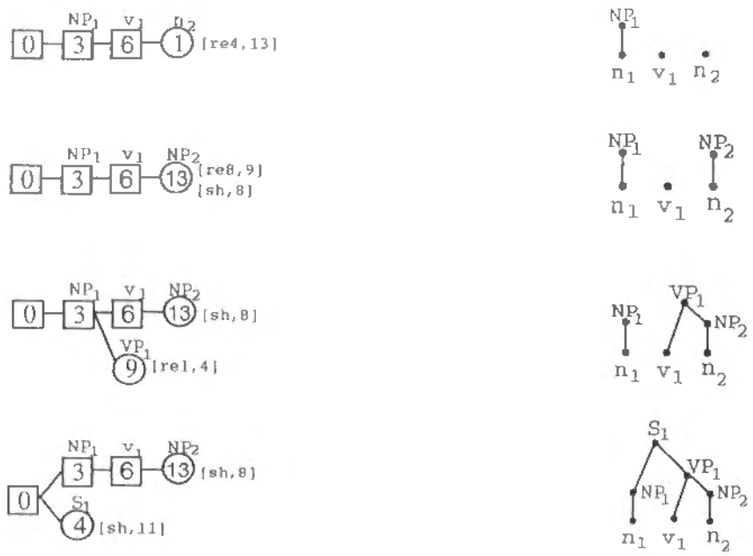


Figure 2.6: Trace of the parser (cont'd)

The two shift actions for the word "and" are then executed, and Figure 2.7 shows the resulting configuration of the parser. The next word is "Jack", which is a noun n. This time, both of the stack tops are active with the same shift-goto pair $[sh, 1]$. This calls for a *combining* of shift nodes on the stack, so the parser generates a single merged node for both shifts. The outcome is depicted in the first row of Figure 2.8 where the next word to be parsed is "hit".

In parsing the word "hit", another instance of graph-structured stack splitting occurs (see rows 1 and 2 of Figure 2.8). In this case, the parser begins with a merged stack top which is active with a *"re 4"* action. Since there are two base nodes (namely, the nodes with states 8 and 11) with different goto states (GOTO[8,*NP*]=16 and GOTO[11, *NP*]=3), the popping of the merged node results in a pair of stack tops. As there is only one

Next word = "Jack" : n

Figure 2.7: Trace of the parser (cont'd)

reduction path, a single parse forest vertex NP_3 is created. Two new stack nodes with states 16 and 3, both sharing NP_3 as their parse forest vertex, are pushed onto the respective base nodes, as shown in the second row of the figure. One of the stack tops (the one with state 16) is active with a reduce action, while the other (the one with state 3) a shift action. Following the policy of GLR, we process the reduce action first. The outcome of this reduction is depicted in the last row of Figure 2.8, where both of the top nodes are active with the same shift-goto pair.

Figure 2.9 shows the result of pushing a combined node onto the stack for the word "hit". The next word is the determiner "the", which calls for a shift action.

Figure 2.10 shows the configuration of the parser after "the" is shifted onto the stack. The action with the next word "man" is again a shift.

In Figures 2.11 and 2.12, the execution sequence of the parser in parsing the preposition "with" is shown.

Let us pay particular attention to the last row of Figure 2.11, especially the active top node with state 9, of which the pending action-goto pair is $[re1, 4]$. The only base node for this reduction is the bottom stack node. However, this base node already has a child node of state 4 which has also beeu created for the current input (from the execution of $[re3, 4]$ on the top node with state 17 in the previous row). This indicates that the current input ("I saw Jane and Jack hit the man") can be reduced into an S in more than one way. Thus, the parser performs *local ambiguity packing* as follows. Instead of creating another stack node and parse forest vertex for the current reduction, the parser packs the new reduction into the parse vertex S_4, causing it to have two sets of children vertices, each corresponding to a possible derivation of S from the input segment "I saw Jane and Jack hit the man" (which can be interpreted as either "[I saw Jane] and [Jack hit

Next word = "hit" : v

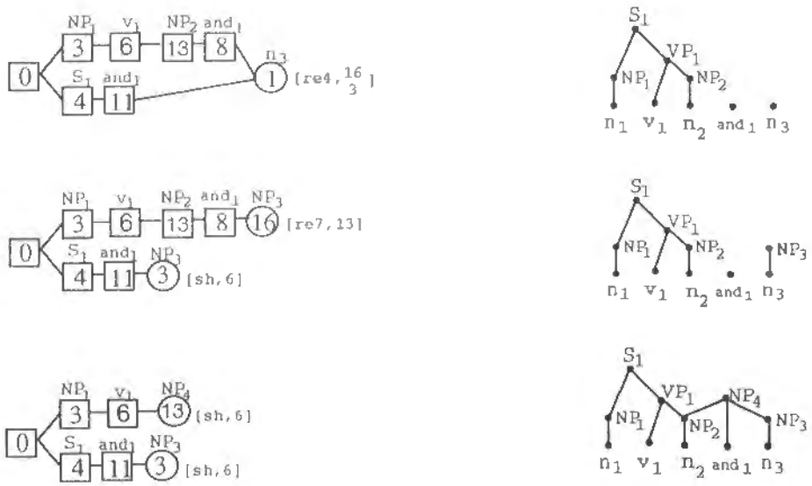


Figure 2.8: Trace of the parser (cont'd)

Next word = "the" : det

Figure 2.9: Trace of the parser (cont'd)

Next word = "man" : n

Figure 2.10: Trace of the parser (cont'd)

Next word = "with":p

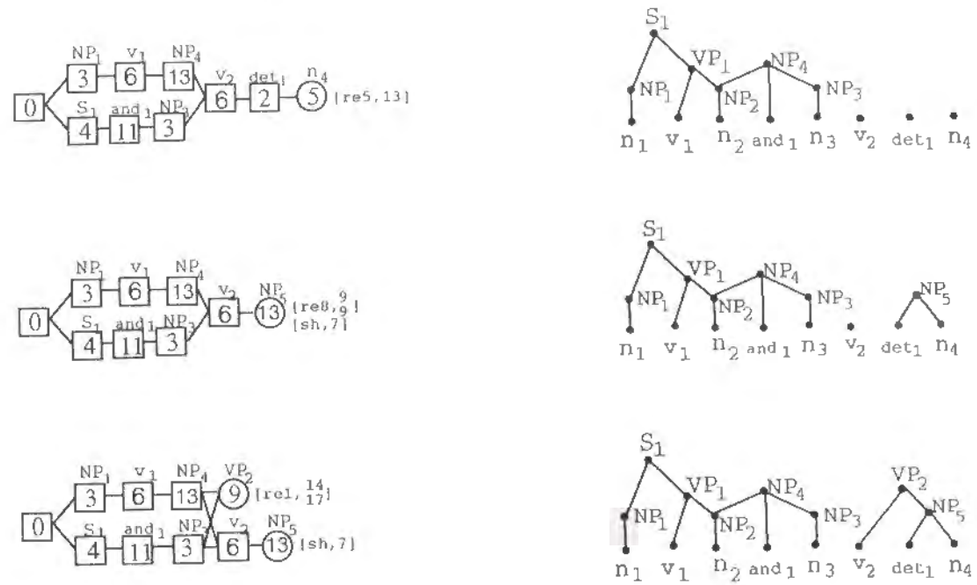
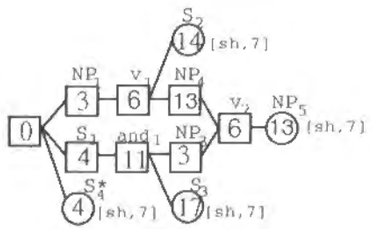


Figure 2.11: Trace of the parser (cont'd)

Next word = "with" : p (cont'd)



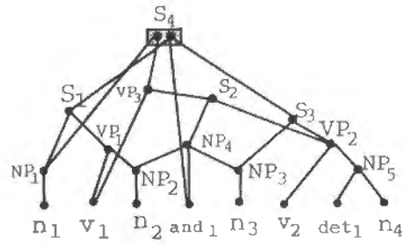


Figure 2.12: Trace of the parser (cont'd)

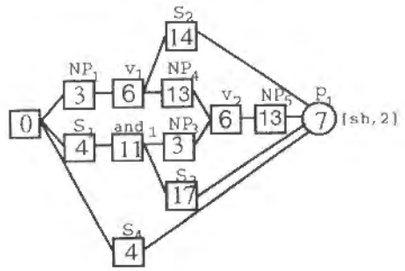
the man]", or "I saw [Jane and Jack hit the man]"). Since this stack node has been created previously by another reduction of the current input, it must be the case that further parsing actions on this node are already taken care of. Thus, the parser does not need to pursue further after packing the parse forest vertex S_3 in this node. Figure 2.12 shows the resulting packed shared parse forest. We indicate where local ambiguity packing occurs on the graph-structured stack by a starred parse vertex (S_4^*). Note also that in Figure 2.12, all the four active stack tops are left with the same shift-goto pair. Again, a combined node for the preposition "with" is created and pushed onto the four tops, the result of which can be seen in Figure 2.13.

Figure 2.13 and 2.14 show the parsing of the last two words in the input sentence, "a" and "telescope", respectively. In each case, a shift action is called to push the word onto the stack.

At this point, the parser has reached the end of the input sentence, so the READ head is looking at the end marker "$". Figures 2.15 and 2.16 shows the sequence of actions taken by the parser towards a final *"accept"* action. The final configuration of the parser is one in which there is only a single active top node on the graph-structured stack, whose only pending action is an *"accept"* action (see the last row of Figure 2.16). The parser thus halts in an accepting state.

The final parse forest and the 6 possible parse trees are shown in Figure 2.17.

Next word = "a": det



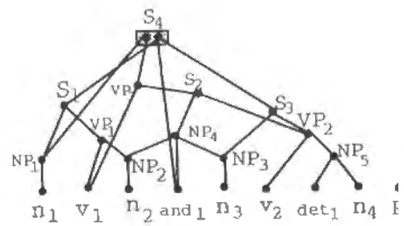
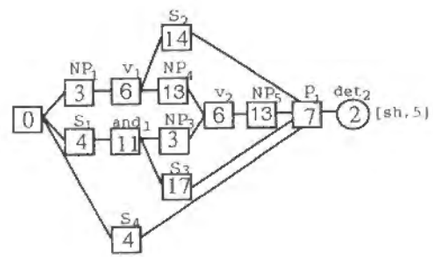


Figure 2.13: Trace of the parser (cont'd)

Next word = "telescope": n



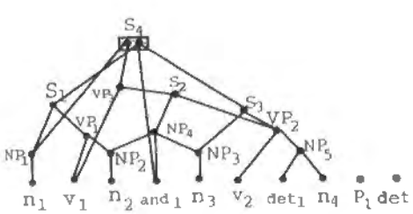


Figure 2.14: Trace of the parser (cont'd)

Next word = "$" : $

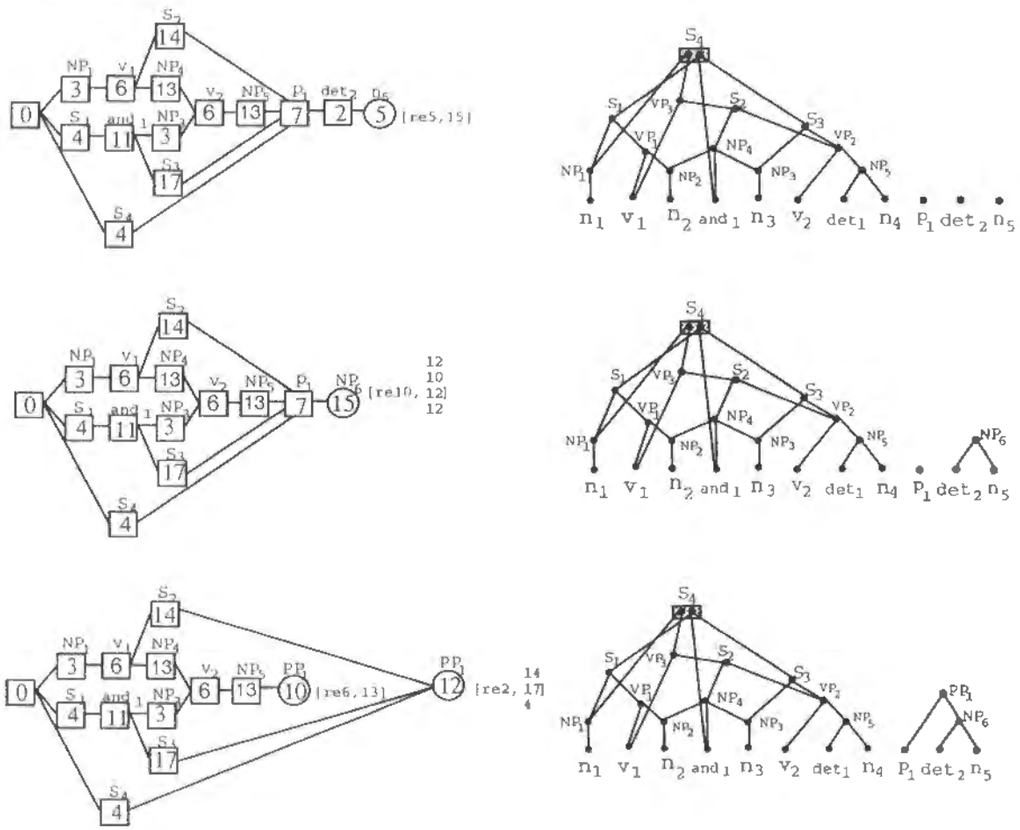


Figure 2.15: Trace of the parser (cont'd)

Next word = "$": $ (cont'd)

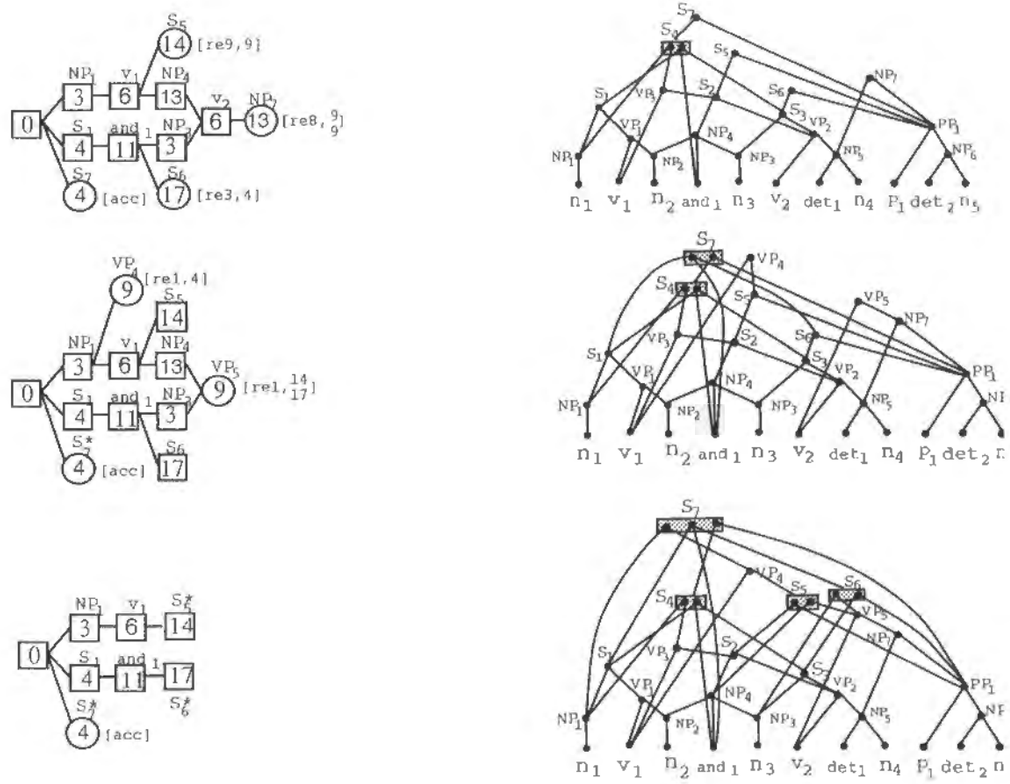
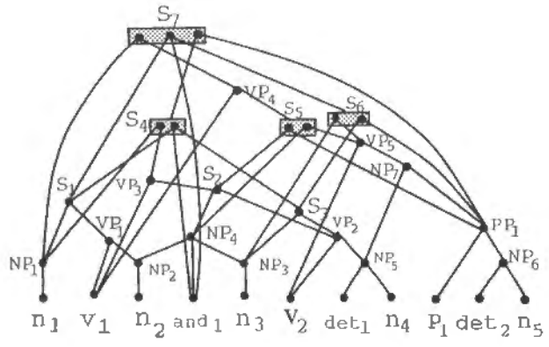


Figure 2.16: Trace of the parser (cont'd)



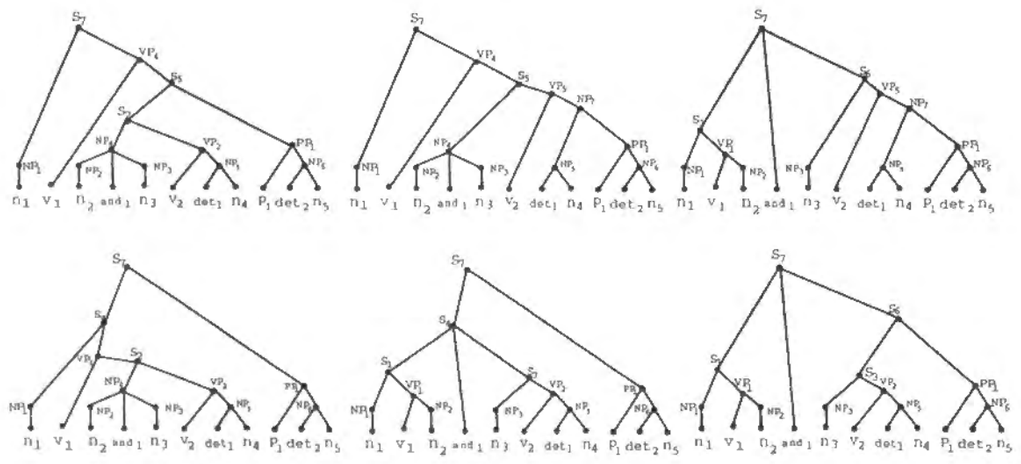


Figure 2.17: Packed Shared Forest and Its Respective Parse Trees

2.5 Specification of the Algorithm

The following is a specification of the GLR(k) parsing algorithm for context-free grammars without ϵ-productions. The algorithm can be easily modified to handle ϵ-productions (see Chapter 5).

The k in this specification refers to the number of input symbols the parser looks ahead during parse time in determining what parsing actions to execute. In our previous example, we used a single lookahead, that is, $k = 1$. Using lookahead strings of size k, the ACTION table is indexed by a state and a string of k terminal symbols, including end-markers \$'s. The GOTO table is indexed by a state and a single grammar symbol. Usually, longer lookaheads are employed during parsing table construction to avoid action conflicts (for example, in canonical LR(k) algorithm or LALR(k) algorithm [Aho and Ullman, 1972, Aho and Ullman, 1977]) so that standard LR parsing can be applied. However, longer lookaheads also result in larger parsing tables. Since generalized LR can handle multiple entries, the choice of k in GLR parsing would depend on the tradeoff between the size of the parsing table and the efficiency obtained from the extra degree of determinism due to longer lookaheads.

Algorithm 2.1 *GLR(k) Parsing Algorithm*

Input: A parsing table for a grammar $G = (N, \Sigma, P, S)$ in terms of an ACTION table which may contain multiple entries, a GOTO function, and an input string $z \in \Sigma^*$. N is the set of nonterminals for G, Σ is the set of terminals, P is the set of productions, and $S \in N$ is the start symbol. ACTION and GOTO are as described in the example in Section 2.4, except that the ACTION table now uses a lookahead string of length k. The state s_0 is designated as the initial state.

Output: If $z \in L(G)$, the root vertex of a packed shared parse forest for z. Otherwise, an error indication.

Method: Patch the input string z with k end markers, giving the string $z\$^k$. Make a stack node η_0 containing the start state s_0. η_0 forms the bottom of the parse stack. Other than η_0, a stack node normally contains two fields: the state of the parser and the corresponding parse forest vertex. Initially, the READ head is pointing at the first symbol of z. Let u denote the lookahead string, which consists of the next k input symbols from the READ head. For each non-error action

$a \in \text{ACTION}[s_0, u]$, add the pair (η_0, a) to an associative list called FRONTIER. The nodes found in the node-action pairs in FRONTIER are the active stack tops. Repeatedly perform step 1, 2, 3 and 4 (in this order) until an acceptance or rejection occurs.

1. Remove an element of the form $(\eta, re\ n)$ from FRONTIER. Let the n-th production be $A \rightarrow \alpha$. Collect P, the set of paths of length $|\alpha|$ ending at η (a path is a contiguous sequence of nodes on the graph). For each path $p \in P$, we create a new parse vertex ν_p to be used in the parse forest as the parent vertex for that reduction. Also, for each p, collect the set of stack nodes which immediately precedes p. We call these nodes the *base nodes* for p, and denote the set as B_p. Partition B_p according to their next goto states on the grammar symbol A. For each goto state s, let b_s be the set of nodes in B_p whose next goto state is s.

   For each set b_s in the partition, check if there is a node η' on the stack which has been created with the current input, whose state is s and children set b_s (consequently, its parse forest vertex also represents the nonterminal A). If so, then local ambiguity has occurred and we can re-use the node η' as the stack node for the current reduction. We pack the parse forest vertex in η' by adding to its children set the corresponding vertex path in p. If no such node is found, then we create a new stack node η'' with s as its state and ν_p as its parse vertex, and make ν_p's children set contain the corresponding vertex path in p. Push η'' onto the stack nodes in b_s, and update the FRONTIER by adding to it a pair (η'', a) for each non-error action $a \in \text{ACTION}[s, u]$.

   Repeat Step 1 until none of the stack tops are active with a reduce action.

2. Remove all pairs from FRONTIER of the form (η, sh). Let the stack nodes in these pairs be $\eta_1, \ldots, \eta_m$, whose states are $s_1, \ldots, s_m$ respectively. Create a new parse vertex ν for u_1, the first symbol in u. Advance the READ head one symbol to the right and let the new lookahead string be w. Let Π be the partition of $\eta_1, \ldots, \eta_m$ according to their next goto states. Let π_s be a set in Π such that $\text{GOTO}[s_i, u_1] = s$ if $\eta_i \in \pi_s$.

   For each $\pi_s \in \Pi$, create a single stack node η_s with state s and parse forest vertex ν, and push η_s onto the stack nodes in π_s.

   Add to FRONTIER a pair (η_s, a) for each non-error action $a \in \text{ACTION}[s, w]$.

3. If FRONTIER $= \{(\eta, acc)\}$, we accept and return the parse vertex in η.
4. If FRONTIER $= \emptyset$, we halt and reject.

2.6 Summary

In this chapter, we saw how standard LR parsing evolves into GLR parsing which handles general context-free grammars instead of LR grammars while retaining much of the efficiency of standard LR parsing. The graph-structured stack and the shared packed parse forest made the efficiency of LR parsing available to natural language processing. Subsequent chapters shall describe extensions of GLR parsing to handle spoken language.

Chapter 3

Parsing with Augmented Grammars

3.1 Introduction

In the previous chapter, we have described the algorithm as a pure context-free parsing algorithm. In practice, it is often desired for each grammar nonterminal to have *attributes*, and for each grammar rule to have an augmentation to define, pass and test the attribute values [1]. It is also desired to produce a functional structure (in the sense of functional grammar formalism [Kay, 1984, Bresnan and Kaplan, 1982], rather than the context-free forest. The subsection 3.1.1 describes the augmentation, and subsection 3.1.2 discusses the shared-packed representation for functional structures.

3.1.1 The Augmentation

We attach a Lisp function to each grammar rule for this augmentation. Whenever the parser reduces constituents into a higher-level nonterminal using a phrase structure rule, the Lisp program associated with the rule is evaluated. The Lisp program handles such aspects as construction of a syntax/semantic representation of the input sentence, passing attribute values among constituents at different levels and checking syntactic/semantic constraints such as subject-verb agreement.

If the Lisp function returns NIL, the parser does not do the reduce action

[1] Major parts of this chapter are based on the author's previously published papers [Tomita, 1987, Tomita, 1990b]

with the rule. If the Lisp function returns a non-NIL value, then this value is given to the newly created non-termiual. The value includes attributes of the nonterminal and a partial syntactic/semantic representation constructed thus far. Notice that those Lisp functions can be precompiled into machine code by the standard Lisp compiler.

3.1.2 Sharing and Packing Functional Structures

A functional structure used in the functional grammar formalisms [Kay, 1984, Bresnan and Kaplan, 1982, Shieber, 1985] is in general a directed acyclic graph (dag) rather than a tree. This is because some value may be shared by two different attributes in the same sentence (e.g. the "agreement" attributes of subject and main verb). Pereira [Pereira, 1985] introduced a method to share dag structures. However, the dag structure sharing method is much more complex and computationally expensive than tree structure sharing. Therefore, we handle only tree-structured functional structures for the sake of efficiency and simplicity [2]. In the example, the "agreement" attributes of subject and main verb may thus have two different values. The identity of these two values is tested explicitly by a test in the augmentation. Sharing tree-structured functional structures requires only a minor modification on the subtree sharing method for the shared-packed forest representation described in section 2.3.

Local ambiguity packing for augmented context-free grammars is not as easy. Suppose certain two nodes have been packed into one packed node. Although these two nodes have the same category name (e.g. NP), they may have different attribute values. When a certain test in the Lisp function refers to an attribute of the packed node, its value may not be uniquely determined. In this case, the parser can no longer treat the packed node as one node, and the parser will unpack the packed node into two individual nodes again. The question, then, is how often this unpacking needs to take place in practice. The more frequently it takes place, the less significant to do local ambiguity packing. However, most of sentence ambiguity comes from such phenomena as PP-attachment and conjunction scoping, and it is unlikely to require unpacking in these cases. For instance, consider the noun phrase:

a man in the park with a telescope,

[2] Although we plan to handle dag structures in the future, tree structures may be adequate, as GPSG use tree structures rather than dag structures.

which is locally ambiguous (whether "telescope" modifies "man" or "park"). Two NP nodes (one for each interpretation) will be packed into one node, but it is unlikely that the two NP nodes have different attribute values which are referred to later by some tests in the augmentation. The same argument holds with the noun phrases:

- pregnant women and children
- large file equipment

Although more comprehensive experiments are desired, it is expected that only a few packed nodes need to be unpacked in practical applications.

3.2 The LFG Compiler

It is in general very painful to create, extend and modify augmentations written in Lisp. The Lisp functions should be generated automatically from more abstract specifications. We have implemented the LFG compiler that compiles augmentations in a higher level notation into Lisp functions. The notation is similar to the Lexical Functional Grammar (LFG) formalism [Bresnan and Kaplan, 1982] and PATR-II [Shieber, 1984]. An example of the LFG-like notation and its compiled Lisp function are shown in figure 3.1 and 3.2. We generate only non-destructive functions with no side-effects to make sure that a process never alters other processes or the parser's control flow. A generated function takes a list of arguments, each of which is a value associated with each right hand side symbol, and returns a value to be associated with the left hand side symbol. Each value is a list of f-structures, in case of disjunction and local ambiguity.

This section describes a software package designed for practical projects which involve natural language parsing. The Generalized LR Parser/Compiler V8-4 is based on the parsing algorithm described in the previous chapter, augmented by pseudo/full unification modules. While the parser/compiler is not a commercial product, it has been thoroughly tested and heavily used by many projects inside and outside CMU last three years. It is publicly available with some restrictions for profit-making industries [3]. It is written

[3]For those interested in obtaining the software, contact Radha Rao, Business Manager, Center for Machine Translation, Carnegie Mellon University, Pittsburgh, PA 15213 (rdr@nl.cs.cmu.edu).


```
(<S>  <==>  (<NP> <VP>)
        (((x1 case) = nom)
         ((x2 form) =c finite)
         (*OR*
           (((x2 :time) = present)
            ((x1 agr) = (x2 agr)))
           (((x2 :time) = past)))
         (x0 = x2)
         ((x0 :mood) = dec)
         ((x0 subj) = x1)))
```

Figure 3.1: Example Grammar Rule in the LFG-like Notation

```
(<S> <==> (<NP> <VP>)
(LAMBDA (X1 X2)
 (LET ((X (LIST (LIST (CONS (QUOTE X2) X2) (CONS (QUOTE X1) X1)))))
  (AND
   (SETQ X (UNIFYSETVALUE* (QUOTE (X1 CASE)) (QUOTE (NOM))))
   (SETQ X (C-UNIFYSETVALUE* (QUOTE (X2 FORM)) (QUOTE (FINITE))))
   (SETQ X (APPEND
            (LET ((X X))
             (SETQ X (UNIFYSETVALUE* (QUOTE (X2 :TIME)) (QUOTE (PRESENT)))))
             (SETQ X (UNIFYVALUE* (QUOTE (X2 AGR)) (QUOTE (X1 AGR))))
             X)
            (LET ((X X))
             (SETQ X (UNIFYSETVALUE* (QUOTE (X2 :TIME)) (QUOTE (PAST))))
             X)))
   (SETQ X (UNIFYVALUE* (QUOTE (X0)) (QUOTE (X2))))
   (SETQ X (UNIFYSETVALUE* (QUOTE (X0 :MOOD)) (QUOTE (DEC))))
   (SETQ X (UNIFYVALUE* (QUOTE (X0 SUBJ)) (QUOTE (X1))))
   (GETVALUE* X (QUOTE (X0)))))))
```

Figure 3.2: The Compiled Grammar Rule

entirely in CommonLisp, and no system-dependent functions, such as window graphics, are used for the sake of portability. Thus, it should run on any systems that run CommonLisp in principle [4], including IBM RT/PC, Mac II, Symbolics and HP Bobcats.

Each rule consists of a context-free phrase structure description and a cluster of *pseudo equations* as in figure 3.1. The non-terminals in the phrase structure part of the rule are referenced in the equations as `x0...xn`, where `x0` is the non-terminal in the left hand side (here, `<DEC>`) and `xn` is the n-th non-terminal in the right hand side (here, `x1` represents `<NP>` and `x2` represents `<VP>`). The pseudo equations are used to check certain attribute values, such as verb form and person agreement, and to construct a f-structure. In the example, the first equation in the example states that the case of `<NP>` must be nominative, and the second equation states that the form of `<VP>` must be finite. Then one of the following two must be true: (1) the time of `<VP>` is present and agreements of `<NP>` and `<VP>` agree, OR (2) the time of `<VP>` is past. If all of the conditions hold, let the f-structure of `<DEC>` be that of `<VP>`, create a slot called "subj" and put the f-structure of `<NP>` there, and create a slot called "passive" and put "-" there. Pseudo equations are described in detail in section 3.4.

Grammar compilation is the key to this efficient parsing system. A grammar written in the correct format is to be compiled before being used to parse sentences. The context-free phrase structure rules are compiled into an *Augmented LR Parsing Table*, and the equations are compiled into CommonLisp functions. The runtime parser then does the shift-reduce parsing guided by the parsing table, and each time a grammar rule is applied, its CommonLisp function compiled from equations is evaluated.

In the subsequence sections, features of the Generalized LR Parser/Compiler v8-4 are briefly described.

3.3 Top-Level Functions

There are three top-level functions:

```
; to compile a grammar (compgra grammar-file-name)
; to load a compiled grammar (loadgra grammar-file-name)
; to parse a sentence string (p sentence)
```

[4] In practice, however, we usually face one or two problems when we transport it to another CommonLisp system, due to bugs in CommonLisp and/or file I/O complications.

3.4 Pseudo Equations

This section describes pseudo equations for the Generalized LR Parser/Compiler V8-4.

3.4.1 Pseudo Unification, =

path = *val*

Get a value from *path*, unify it with *val*, and assign the unified value back to *path*. If the unification fails, this equation fails. If the value of *path* is undefined, this equation behaves like a simple assignment. If *path* has a value, then this equation behaves like a test statement.

path1 = *path2*

Get values from *path1* and *path2*, unify them, and assign the unified value back to *path1* and *path2*. If the unification fails, this equation fails. If both *path1* and *path2* have a value, then this equation behaves like a test statement. If the value of *path1* is not defined, this equation behaves like a simple assignment.

3.4.2 Overwrite Assignment, <=

path <= *val*

Assign *val* to the slot *path*. If *path1* is already defined, the old value is simply overwritten.

path1 <= *path2*

Get a value from *path2*, and assign the value to *path1*. If *path1* is already defined, the old value is simply overwritten.

path <= *lisp-function-call*

Evaluate *lisp-function-call*, and assign the returned value to *path*. If *path1* is already defined, the old value is simply overwritten. *lisp-function-call* can be an arbitrary lisp code, as long as all functions called in *lisp-function-call* are defined. A path can be used as a special function that returns a value of the slot.

3.4.3 Removal Assignment, ==

path1 == *path2*

Get a value from *path2*, assign the value to *path1*, and remove the value of *path2* (assign nil to *path2*). If a value already exists in *path1*, then the new value is unified with the old value. If the unification fails, then this equation fails.

3.4.4 Append Multiple Value, >

path1 > *path2*

Get a value from *path2*, and assign the value to *path1*. If a value already exists in *path1*, the new value is appended to the old value. The resulting value of *path1* is a multiple value.

3.4.5 Pop Multiple Value, <

path1 < *path2*

The value of *path2* should be a multiple value. The first element of the multiple value is popped off, and assign the value to *path1*. If *path1* already has a value, unify the new value with the old value. If *path2* is undefined, this equation fails.

3.4.6 *DEFINED* and *UNDEFINED*

path = *DEFINED*

Check if the value of *path* is defined. If undefined, then this equation fails. If defined, do nothing.

3.4.7 Constraint Equations, =c

path =c *val*

This equation is the same as an equation

path = *val*

except if *path* is not already defined, it fails.

3.4.8 Removing Values, *REMOVE*

path = *REMOVE*

This equation removes the value in *path*, and the path becomes undefined.

3.4.9 Disjunctive Equations, *OR*

(*OR* *list-of-equations list-of-equations* ....)

All lists of equations are evaluated disjunctively. This is an inclusive OR, as oppose to exclusive OR; Even if one of the lists of equations is evaluated successfully, the rest of lists will be also evaluated anyway.

3.4.10 Exclusive OR, *EOR*

(*EOR* *list-of-equations list-of-equations* ....)

This is the same as disjunctive equations *OR*, except an exclusive OR is used. That is, as soon as one of the element is evaluated successfully, the rest of elements will be ignored.

3.4.11 Case Statement, *CASE*

(*CASE* *path* (*key1 equation1-1 equation1-2 ...*) (*Key2 equation2-1 ...*) (*Key3 equation3-1 ...*) ....)

The *CASE* statement first gets the value in *path*. The value is then compared with Key1, Key2, ..., and as soon as the value is eq to some key, its rest of equations are evaluated.

3.4.12 Test with an User-defined LISP Function, *TEST*

(*TEST* *lisp-function-call*)

The *lisp-function-call* is evaluated, and if the function returns nil, it fails. If the function returns a non-nil value, do nothing. A path can be used as special function that returns a value of the slot.

3.4.13 Recursive Evaluation of Equations, *INTERPRET*

(*INTERPRET *path*)

The *INTERPRET* statement first gets a value from *path*. The value of *path* must be a valid list of equations. Those equations are then recursively evaluated. This *INTERPRET* statement resembles the "eval" function in Lisp.

3.4.14 Disjunctive Value, *OR*

(*OR* *val val ...*)

Unification of two disjunctive values is set interaction. For example, `(unify '(*OR* a b c d) '(*OR* b d e f))` is `(*OR* b d)`.

3.4.15 Negative Value, *NOT*

(*NOT* *val val ...*)

Unification of two negative values is set union. For example, `(unify '(*NOT* a b c d) '(*NOT* b d e f))` is `(*NOT* a b c d e f)`.

3.4.16 Multiple Values, *MULTIPLE*

(*MULTIPLE* *val val ...*)

Unification of two multiple values is append. When unified with a value, each element is unified with a value. For example, `(unify '(*MULTIPLE* a b c d b d e f) 'd)` is `(*MULTIPLE* d d)`.

3.4.17 User Defined special Values, **user-defined**

The user can define his own special values. An unification function with the name UNIFY**user-defined** must be defined. The function should take two arguments, and returns a new value or *FAIL* if the unification fails.

3.5 Standard Unification Mode

The pseudo equations described in the previous section are different from what functional grammarians call "unification". The user can, however, select "full (standard) unification mode" by setting the global variable *UNIFICATION-NODE* from PSEUDO to FULL. In the full unification mode, equations are interpreted as standard equations in a standard functional unification grammar [Shieber, 1986], although some of the features such as user-defined function calls cannot be used. However, most users of the parser/compiler find it more convenient to use PSEUDO unification than FULL unification, bot only because it is more efficient, but also because it has more practical features including user-defined function calls and user-defined special values. Those practical features are crncial to haudle low-level non-linguistic phenomena such as time and date expressions [Tomita, 1988c] and/or to incorporate semantic and pragmatic processing of the user's choice.

3.6 Other Important Features

3.6.1 Character Basis Parsing

The user has a choice to make his grammar "character basis" or standard "word basis". When "character basis mode" is chosen, terminal symbols in the grammar are characters, not words. There are at least two possible reasons to make it character basis:

1. Some languages, such as Japanese, do not have a space between words. If a grammar is written in character basis, the user does not have to worry about word segmentation of unsegmented sentences.

2. Some languages have mnch more complex morphology than English. With the character basis mode, the user can write morphological rules in the very same formalism as syntactic rules.

3.6.2 Wild Card Character

In pseudo unification mode, the user can use a wild card character "character (if character basis) or any word (if word basis). This feature is especially useful to handle proper nouns and/or unknown words.

3.6.3 Grammar Debugging Tools

The Generalized LR Parser/Compiler V8-4 includes some debugging fnnctions. They include:

- `dmode` — debugging mode; to show a trace of rule applications by the parser.
- `trace` — to trace a particular rule.
- `disp-trees, disp-nodes, etc.` — to display partial trees or values of nodes in a tree.

All of the debugging tools do not use any fancy graphic interface for the sake of system portability.

3.6.4 Interpretive Parser

The Generalized LR Parser/Compiler V8-4 includes another parser based on chart parsing which can parse a sentence without ever compiling a grammar:

; to load a grammar (`i-loadgra` *grammar-file-name*)
; to run the interpretive parser (`i-p` *sentence*)

While its run time speed is significantly slower than that of the GLR parser, many users find it useful for debugging because grammar does not need to be compiled each time a small change is made.

3.6.5 Grammar Macros

The user can define and use macros in a grammar. This is especially useful in case there are many similar rules in the grammar. A macro can be defined in the same way as CommonLisp macros. Those macros are expanded before the grammar is compiled.

3.7 Summary

Some of the important features of the Generalized LR Parser/Compiler have been highlighted. More detailed descriptions can be found in Appendix A.

Unlike most other available software [Karttunen, 1986, Kiparsky, 1985, Shieber, 1984], the Generalized LR Parser/Compiler v8-4 is designed specifically to be used in practical natural language systems, sacrificing perhaps

some of the linguistic and theoretical elegancy. The system has been thoroughly tested and heavily used by many users in many projects world wird since 1988. Center for Machine Translation of Carnegie Mellon University has developed rather extensive grammars for English and Japanese for their translation projects, and some experimental grammars for French, Spanish, Turkish and Chinese. We also find the system very suitable to write and parse task-dependent semantic grammars.

Chapter 4

Parsing with Probabilistic Grammars

4.1 Introduction

Probabilistic grammars provide a formalism which accounts for certain statistical aspects of the language, allows stochastic disambiguation of sentences [1]. As described in chapter 2, Generalized LR parsing is a highly efficient parsing algorithm that has been adapted to handle arbitrary context-free grammars. To combine the advantages of both mechanisms, an algorithm for constructing a generalized probabilistic LR parser given a probabilistic context-free grammar is needed. In Wright and Wrigley [Wright and Wrigley, 1989], a probabilistic LR-table construction method has been proposed for non-left-recursive context-free grammars. However, in practice, left-recursive context-free grammars are not uncommon, and it is often necessary to retain this left-recursive grammar structure. Thus, a method for handling left-recursions is needed in order to attain probabilistic LR-table construction for *general* context free grammars.

In this chapter, we concentrate on incorporating probabilistic grammars with generalized LR parsing for efficiency. Stochastic information from probabilistic grammar can be used in making statistical decision during runtime to improve performance. In Section 4.3, we show how to adapt the generalized LR parser with graph-structured stack to perform probabilistic parsing

[1] Major parts of this chapter are based on the author's previously published paper [Ng and Tomita, 1991]. I would like to acknowledge the coauther of the paper, See-Kiong Ng, whose contribution is included in this chapter.

and discuss related implementation issues. In Section 4.4, we describe the difficulty in computing item probabilities for left recursive context-free grammars. A solution is proposed in Section 4.5, which involves encoding item dependencies in terms of a system of linear equations. These equations can then be solved by Gaussian Elimination [Strang, 1980] to give the item probabilities, from which the stochastic factors of the corresponding parse actions can be computed as described in Wright and Wrigley [Wright and Wrigley, 1989].

We also introduce the notion of *deferred probability* in Section 4.6 in order to prevent creating excessive number of duplicate items which are similar except for their probability assignments.

4.2 Background

Probabilistic LR parsing is based on the notions of probabilistic context-free grammar and probabilistic LR parsing table, which are both augmented versions of their nonprobabilistic counterparts. In this section, we provide the definitions for the probabilistic versions.

4.2.1 Probabilistic Context-Free Grammar

A *probabilistic context-free grammar* (PCFG) [Suppes, 1970, Wetherall, 1980, Wright and Wrigley, 1989] G, is a 4-tuple $\langle N, T, R, S \rangle$ where N is a set of non-terminal symbols including S the start symbol, T a set of terminal symbols, and R a set of probabilistic productions of the form $< A \rightarrow \alpha,\ p >$ where $A \in N$, $\alpha \in (N \cup T)^*$, and p the production probability. The probability p is the conditional probability $P(\alpha|A)$, which is the probability that the non-terminal A which appears during a derivation process is rewritten by the sequence α. Clearly if there are k A-productions with probabilities $p_1, \ldots, p_k$, then $\sum_{i=1}^{k} p_i = 1$, since the symbol A must be rewritten by the right hand side of some A-production. The production probabilities can be estimated from the corpus as outlined in Fu and Booth [Fu and Booth, 1975] or Fujisaki [Fujisaki, 1984].

It is assumed that the steps of every derivation in the PCFG are mutually independent, meaning that the probability of applying a rewrite rule depends only upon the presence of a given nonterminal symbol (the premis) in a derivation and not upon how the premis was generated. Thus, the probability of a derivation is simply the product of the production probabilities of the productions in the derivation sequence.

(1)	$S \rightarrow NP\ VP$	1
(2)	$NP \rightarrow n$	$\frac{1}{3}$
(3)	$NP \rightarrow det\ n$	$\frac{2}{3}$
(4)	$VP \rightarrow v\ NP$	1

Figure 4.1: GRA1: A Non-left Recursive PCFG

(1)	$S \rightarrow NP\ VP$	$\frac{3}{4}$
(2)	$S \rightarrow S\ PP$	$\frac{1}{4}$
(3)	$NP \rightarrow n$	$\frac{1}{2}$
(4)	$NP \rightarrow det\ n$	$\frac{2}{5}$
(5)	$NP \rightarrow NP\ PP$	$\frac{1}{10}$
(6)	$PP \rightarrow prep\ NP$	1
(7)	$VP \rightarrow v\ NP$	1

Figure 4.2: GRA2: A Left-recursive PCFG

Figures 4.1, 4.2 and 4.3 show three example PCFGs GRA1, GRA2 and GRA3 respectively. Incidentally, GRA1 is non-left recursive, GRA2 and GRA3 are both left-recursive, although GRA3 is "more" left-recursive than GRA2. GRA2 is said to have *simple recursion* since there is only a finite number of distinct left-recursive loops[2] in the grammar. GRA3, on the other hand, is said to have *massive left recursions* because of the intermingled left recursions, which result in infinite (possibly uncountable) number of distinct left-recursive loops in the grammar.

4.2.2 Probabilistic LR Parse Table

A probabilistic LR table is an augmented LR table of which the entries in the ACTION-table contains an additional field which is the probability of the action. We call this probability *stochastic factor* because it is the factor used in the computation (multiplication) of the *runtime stochastic product*. The parser keeps this stochastic product during runtime for each possible derivation, reflecting their respective likelihoods. This product can be computed during runtime by multiplication using the precomputed stochastic factors of the parsing actions (or by addition if the stochastic factors are expressed in logarithms). The parser can use this stochastic information to disambiguate or direct/prune its search probabilistically.

[2] A loop is a derivation cycle in which the first and last productions used in the derivation sequence are the same and occur nowhere else in the sequence.

(1)	$S \rightarrow S\ a_1$	$\frac{1}{7}$
(2)	$S \rightarrow B\ a_2$	$\frac{4}{7}$
(3)	$S \rightarrow C\ a_3$	$\frac{2}{7}$
(4)	$B \rightarrow S\ a_3$	$\frac{1}{3}$
(5)	$B \rightarrow B\ a_2$	$\frac{1}{6}$
(6)	$B \rightarrow C\ a_1$	$\frac{1}{2}$
(7)	$C \rightarrow S\ a_2$	$\frac{1}{5}$
(8)	$C \rightarrow B\ a_3$	$\frac{4}{15}$
(9)	$C \rightarrow C\ a_1$	$\frac{1}{15}$
(10)	$C \rightarrow a_3\ B$	$\frac{1}{3}$
(11)	$C \rightarrow a_3$	$\frac{2}{15}$

Figure 4.3: GRA3:A Massively Left-recursive PCFG

Figures 4.4, 4.5 and 4.6 show the respective probabilistic parsing tables for GRA1, GRA2 and GRA3, as constructed by the algorithm outlined in Section 4.5. Note that the stochastic factors of distinct actions associated with a state add up to 1 as expected, since each action's stochastic factor is simply the probability of the parser making that action during that point of parse. The format of the GOTO-table is unchanged as no stochastic factor is associated with GOTO actions.

4.3 Probabilistic Generalized LR Parsing

In this section, we describe how the efficient generalized LR parser with graph-structured stack can be adapted to parse probabilistically using the augmented parsing table. In particular, we discuss how to maintain consistent runtime stochastic products base on three key notions of the graph-structured stack: merging, local ambiguity packing and splitting. We assume that the state number and the respective runtime stochastic product are stored at each stack node.

State	ACTION				GOTO		
	det	*n*	*v*	$	*NP*	*VP*	*S*
0	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			4		3
1			$\langle re2, 1\rangle$	$\langle re2, 1\rangle$			
2		$\langle sh5, 1\rangle$					
3				$\langle acc, 1\rangle$			
4			$\langle sh6, 1\rangle$			7	
5			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
6	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			8		
7				$\langle re1, 1\rangle$			
8				$\langle re4, 1\rangle$			

Figure 4.4: Probabilistic Parsing Table for GRA1

State	ACTION					GOTO			
	det	*n*	*v*	*prep*	$	*NP*	*PP*	*VP*	*S*
0	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				3			4
1			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$				
2		$\langle sh5, 1\rangle$							
3			$\langle sh7, \frac{9}{10}\rangle$	$\langle sh6, \frac{1}{10}\rangle$			8	9	
4				$\langle sh6, \frac{1}{4}\rangle$	$\langle acc, \frac{3}{4}\rangle$		10		
5			$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
6	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				11			
7	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				12			
8			$\langle re5, 1\rangle$	$\langle re5, 1\rangle$	$\langle re5, 1\rangle$				
9				$\langle re1, 1\rangle$	$\langle re1, 1\rangle$				
10				$\langle re2, 1\rangle$	$\langle re2, 1\rangle$				
11			$\langle re6, \frac{9}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$		8		
12			$\langle re7, \frac{9}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$		8		

Figure 4.5: Probabilistic Parsing Table for GRA2

State	ACTION				GOTO		
	a_1	a_2	a_3	$	*S*	*B*	*C*
0			$\langle sh1, 1\rangle$		2	3	4
1	$\langle re11, \frac{2}{7}\rangle$		$\langle re11, \frac{2}{7}\rangle$ $\langle sh1, \frac{5}{7}\rangle$		5	6	7
2	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{33}{203}\rangle$	$\langle sh10, \frac{64}{203}\rangle$	$\langle acc, \frac{11}{29}\rangle$			
3		$\langle sh11, \frac{37}{48}\rangle$	$\langle sh12, \frac{11}{48}\rangle$				
4	$\langle sh13, \frac{107}{165}\rangle$		$\langle sh14, \frac{58}{165}\rangle$				
5	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{66}{259}\rangle$	$\langle sh10, \frac{156}{259}\rangle$				
6	$\langle re10, \frac{77}{234}\rangle$	$\langle sh15, \frac{113}{234}\rangle$	$\langle re10, \frac{77}{234}\rangle$ $\langle sh12, \frac{22}{117}\rangle$				
7	$\langle sh16, \frac{128}{165}\rangle$		$\langle sh14, \frac{37}{165}\rangle$				
8	$\langle re7, 1\rangle$		$\langle re7, 1\rangle$				
9	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$			
10	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
11	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$			
12	$\langle re8, 1\rangle$		$\langle re8, 1\rangle$				
13	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$				
14	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
15	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$			
16	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$				

Figure 4.6: Probabilistic Parsing Table for GRA3

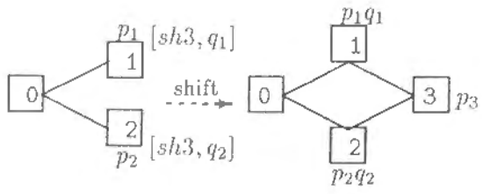


Figure 4.7: Merging

4.3.1 Merging

Merging occurs when an element is being shifted onto two or more of the stack tops. Figure 4.7 illustrates a typical scenario in which a new state (State 3) is pushed onto stack tops States 1 and 2, of which original stochastic products are p_1 and p_2 respectively. These two nodes's stochastic products are modified to p_1q_1 and p_2q_2 correspondingly. If the stochastic factors of the actions has been represented as logarithms in the parse table, then their new "product" (or rather, logarithmic sums) would be $p_1 + q_1$ and $p_2 + q_2$ instead. For the stochastic product of Node 3, we can either use the sum of its parents' products (giving p_3 as $p_1q_1 + p_2q_2$) if we adopt *strict probabilistic approach*, or the maximum of the products (ie, $p_3 = \max(p_1q_1, p_2q_2)$) if we adopt the *maximum likelihood approach*. Note that although the maximum likelihood approach is in some sense less "accurate" than the strict probabilistic approach, it is a reasonable approximate and has an added advantage when the stochastic factors are represented in logarithms, in which case the stochastic "products" of the parse stack can be maintained using only addition and subtraction operators(assuming, of course, that additions and subtractions are "cheaper" computationally than multiplications and divisions).

4.3.2 Local Ambiguity Packing

Local ambiguity packing occurs when two or more branches of the stack are reduced to the same nonterminal symbol. To be precise, this occurs when the parser attempts to create a GOTO state node (after a reduce action, that is) and realize that the parent already has a child node of the same state. In this case there is no need to create the GOTO node but to use that

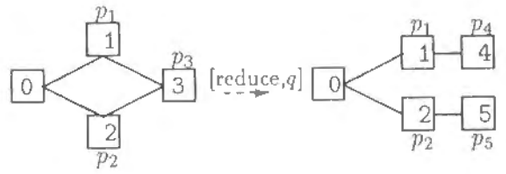


Figure 4.8: Splitting

child node ("packing"). This is equivalent to the merging of shift nodes, and can be handled similarly: the runtime product of the child node is modified to the new "merged" product (either by summation or maximalization). This modification should be propagated accordingly to the successors of the packed child node, if any.

4.3.3 Splitting

Splitting occurs when there is an action conflict. This can be handled straightforwardly by creating corresponding new nodes for the new resulting states with the respective runtime products (such as the product of the parent's stochastic product with the action's stochastic factor). Splitting can also occur when reducing (popping) a merged node. In this case, the parser needs to recover the original runtime product of the merged components, which can be obtained with some mathematical manipulation from the runtime products recorded in the merged node's parents. Figure 4.8 illustrates a simple situation in which a merged node is split into two. In the figure, a reduce action (of which the corresponding production is of unit length) is applied at Node 3, and the GOTO's for Nodes 1 and 2 are states 4 and 5 respectively. In the case that strict probabilistic approach is used in merging (see above), we get $p_4 = \frac{p_1}{p_1+p_2}p_3q$ and $p_5 = \frac{p_2}{p_1+p_2}p_3q$. If the maximum likelihood approach is used, then $p_4 = \frac{p_1}{\max(p_1,p_2)}p_3q$ and $p_5 = \frac{p_2}{\max(p_1,p_2)}p_3q$. Furthermore, if the stochastic factors have been expressed in logarithms, then $p_4 = p_3 - \max(p_1, p_2) + p_1 + q$ and $p_5 = p_3 - \max(p_1, p_2) + p_2 + q$ (notice that only addition and subtraction are needed, as promised).

In general, there may be more than one splitting corresponding to a reduce action (ie, we may have to pop more than one merged nodes). For every split node, we must recover the runtime products of its parents to

obtain the appropriate stochastic products for the resulting new branches. This can be tricky and is one of the reasons why a tree-structured stack (described below) instead of graphs might perform better in some cases.

4.3.4 Using Stochastic Product to Guide Search

The main point of maintaining the runtime stochastic products is to use it as a good indicator function to guide search. In practical situation, the grammar can be highly ambiguous, resulting in many branches of ambiguity in the parse stack. As discussed before, the runtime stochastic product reflects the likelihood of that branch to complete successfully.

In the generalized LR parser, processes are synchronized by performing all the reduce actions before the shift actions. In this way, the processes are made to scan the input at the same rate, which in turn allows the unification of processes in the same state. Thus, the runtime stochastic products can be a good enough indicator of how promising each branch (ie. partial derivation) is, since we are comparing among partial derivations of same input length. We can perform beam search by pruning away branches which are less promising.

If instead of the breadth-first style beam search approach described above we employ a best-first (or depth-first) strategy, then not all of the branches will correspond to the same input length. Since the measure of runtime stochastic product is biased towards shorter sentences, a good heuristic would have to take into account of the number of input symbols consumed. Even so, handling best-first search can be tricky with the graph-structured stack without the process-input synchronization, especially with the merging and packing of nodes. Presumably, we can have additional data structure to serve as lookup table of the nodes currently in the graph stack: for instance, an n by m matrix (where n is the number of states in the parse table and m the input length) indexed by the state number and the input position storing pointers to current stack nodes. With this lookup table, the parser can check if there is any stack node it can use before creating a new one. However, in the worst case, the nodes that could have been merged or packed might have already been popped of the stack before it can be re-used. In this case, the parser degenerates into one with tree-structured stack (ie, only splitting, but no merging and packing) and the laborious book-keeping of the stochastic products due to the graph structure of the parse stack seems wasted. It might be more productive then to employ a tree-structured stack instead of a graph-structured stack, since the book-keeping of runtime stochastic products for trees is much simpler: as each tree branch represents exactly one possible parse, we can associate the respective runtime stochastic products to the leaf nodes (instead of every node) in the parse stack, and updating would involve only multiplying (or adding, in the logarithmic case) with the stochastic factors of the corresponding parse actions to obtain the new stochastic products. The major drawback of the tree-stack version is that it is merely a slightly compacted form of stack list [Tomita, 1987] — which means that the tree can grow unmanageably large in a short period, unless suitable pruning is done. Hopefully, the runtime stochastic product will serve as good heuristic for pruning the branches; but whether it is the case that the simplicity of the tree implementation overrides that of the representational efficiency of the graph version remains to be studied.

4.4 Problem with Left Recursion

The approach to probabilistic LR table construction for non-left recursive PCFG , as proposed by Wright and Wrigley [Wright and Wrigley, 1989], is to augment the standard SLR table construction algorithm presented in Aho and Ullman [Aho and Ullman, 1977] to generate a probabilistic version. The notion of a probabilistic item $\langle A \rightarrow \alpha \cdot \beta,\ p \rangle$ is introduced, with $\langle A \rightarrow \alpha \cdot \beta \rangle$ being an ordinary LR(0) item, aud p the item probability, which is interpreted as the posterior probability of the item in the state. The major extension is the computation of these item probabilities from which the stochastic factors of the parse actions can be determined. Wright and Wrigley [Wright and Wrigley, 1989] have shown a direct method for computing the item probabilities for non-left recursive grammars. The probabilistic parsing table in Figure 4.4 for the non-left recursive grammar GRA1 is thus constructed.

Since there is an algorithm for removing left recursions from a context-free grammar [Aho and Ullman, 1977], it is conceivable that the algorithm can be modified to convert a left-recursive PCFG to one that is non left-recursive. Given a left-recursive PCFG, we can apply this algorithm, and theu use Wright and Wrigley's table construction method [Wright and Wrigley, 1989] on the resulting non left-recursive grammar to create the parsing table. Unfortunately, the left-recursion elimination algorithm destructs the original grammar structure. In practice, especially in natural language processing, it is often necessary to preserve the original grammar structure. Hence a method for constructing a parse table without grammar conversion

I_0:	$[VP \rightarrow v \cdot NP,$	$S_0]$
I_1:	$[NP \rightarrow \cdot n,$	$S_1]$
I_2:	$[NP \rightarrow \cdot det\ n,$	$S_2]$
I_3:	$[NP \rightarrow \cdot NP\ PP,$	$S_3]$

Figure 4.9: An Example State for GRA2

is needed.

For grammars with left recursion, the computation of item probabilities becomes nontrivial. First of all, item probability ceases to be a "probability", as an item which is involved in left recursion is effectively a coalescence of an infinite number of similar items along the cyclic paths, so its associated stochastic value is the sum of posteriori probabilities of these packed items. For instance, if starting from item $\langle A \rightarrow \alpha \cdot B\beta,\ p\rangle$ we derive the item $\langle C \rightarrow \cdot B\gamma,\ p \times p_B\rangle$, then by left recnrsion we must also have the items $\langle C \rightarrow \cdot B\gamma,\ p \times p_B^i\rangle$ for $i = 1, \ldots \infty$. The probabilistic item $\langle C \rightarrow \cdot B\gamma,\ q\rangle$, being a coalescence of these items, would have item probability $q = \sum_{i=1}^{\infty} p \times p_B^i = \frac{p}{1-p_B}$, and there is no guarantee that $q \leq 1$. This is understandable since $\langle C \rightarrow \cdot B\gamma, q\rangle$ is a coalescence of items which are not necessarily mutually exclusive. However, we need not be alarmed as the stochastic values of the underlying items are still legitimate probabilities.

Owing to this coalescence of infinite items into one single item in left recursive grammars, the computation of the stochastic values of items involves finding infinite sums of the items' stochastic values. For grammars with simple left recursion (that is, there are only finitely many left recursion loops) such as GRA2, we can still figure out the sum by enumeration, since there is only a finite number of the infinite sums corresponding to the left recursion loops. With massive left recursive grammars like GRA3 in which there is an infinite number of (intermingled) left recursion loops, the enumeration method fails. We shall illustrate this effect in the following sections.

4.4.1 Simple Left Recursion

For grammars with simple left recursion, it is possible to derive the stochastic values by simple cycle detection. For instance, consider the following set of LR(0) items for GRA2 in Figure 4.9.

Suppose the kernel set contains only I_0, with $S_0 = \frac{3}{7}$. Let $\mathcal{D}$ be a partial

I_0:	$[S' \rightarrow \cdot S,$	$1]$
I_1:	$[S \rightarrow \cdot Sa_1,$	$S_1]$
I_2:	$[S \rightarrow \cdot Ba_2,$	$S_2]$
I_3:	$[S \rightarrow \cdot Ca_3,$	$S_3]$
I_4:	$[B \rightarrow \cdot Sa_3,$	$S_4]$
I_5:	$[B \rightarrow \cdot Ba_2,$	$S_5]$
I_6:	$[B \rightarrow \cdot Ca_1,$	$S_6]$
I_7:	$[C \rightarrow \cdot Sa_2,$	$S_7]$
I_8:	$[C \rightarrow \cdot Ba_3,$	$S_8]$
I_9:	$[C \rightarrow \cdot Ca_1,$	$S_9]$
I_{10}:	$[C \rightarrow \cdot a_3 B,$	$S_{10}]$
I_{11}:	$[C \rightarrow \cdot a_3,$	$S_{11}]$

Figure 4.10: Start State of GRA3

derivation before seeing the input symbol v. At this point, the possible derivations which will lead to item I_1 are:

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{10}}{\Longrightarrow} NP \rightarrow \cdot NP\ VP \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\vdots$

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{10}}{\Longrightarrow} NP \rightarrow \cdot NP\ VP \overset{\frac{1}{10}}{\Longrightarrow} \ldots \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\vdots$

The sum of the posterior probabilities of the above possible partial derivations are:

$S_1 = (S_0 \times \frac{1}{2}) + (S_0 \times \frac{1}{10} \times \frac{1}{2}) + (S_0 \times \frac{1}{10}^2 \times \frac{1}{2}) + \cdots$

$= \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{1}{2} = \frac{5}{21}$

Similarly, $S_2 = \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{2}{5} = \frac{4}{21}$, and $S_3 = \frac{3}{7} \times \sum_{n=1}^{\infty} \frac{1}{10}^n = \frac{1}{21}$.

4.4.2 Massive Left Recursion

For grammars with interningled left recursions such as GRA3, computation of the stochastic values of the items becomes a convoluted task. Consider the start state for GRA3, which is depicted in Figure 4.10.

Consider the item I_1. In an attempt to write down a closed expression for the stochastic value S_1, we discover in despair that there is an infinite number of loops to detect, as S is immediately reachable by all non-terminals, and

so are the other nonterminals themselves. This intermingling of the loops renders it impossible to write down closed expressions for S_1 through S_{11}.

4.5 Construction of Probabilistic LR Parsing Table

In this section, we describe a way of computing item probabilities by encoding the item dependencies in terms of systems of linear equations and solving them by Gaussian Elimination [Strang, 1980]. This method handles arbitrary context-free grammar including those with left recursions. We incorporate this method with Wright and Wrigley's algorithm [Wright and Wrigley, 1989] for computing stochastic factors for the parse actions to obtain a table construction algorithm which handles general PCFG. A formal description of the complete table construction algorithm is in the Appendix.

In the following discussion of the algorithm, lower case greek characters such as α and β will denote strings in $(N \cup T)^*$ and upper case alphabets like A and B denote symbols in N unless mentioned otherwise.

4.5.1 Stochastic Values of Kernel Items

For completeness, we mention briefly here how the stochastic values of items in the kernel set can be computed as proposed by Wright and Wrigley [Wright and Wrigley, 1989]:

The stochastic value of the kernel item $[S' \rightarrow \cdot S]$ in the start state is 1. Let State $m-1$ be a prior state of the non-start State m. We want to compute the stochastic values of the kernel items of State m. Suppose in State $m-1$ there are k items which are expecting the grammar symbol X, their stochastic values being $S_1, S_2, \ldots, S_k$ respectively. Let $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i]$ be these item, $i = 1, \ldots, k$. Then the posterior probability of the kernel item $[A_i \rightarrow \alpha_i X \cdot \beta_i]$ of State m given those k items in State i and grammar symbol X as the next symbol seen on the parse stack is $\frac{S_i}{S_X}$, where $S_X = \sum_{i=1}^{k} S_i$.

4.5.2 Dependency Graph

The inter-dependency of items within a state can be represented most straightforwardly by a dependency forest. If we label each arc by the probability of the rule represented by that item the arc is pointing at, then the posterior

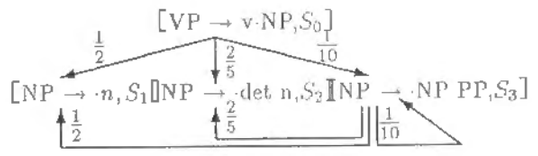


Figure 4.11: A Dependency Graph

probability of an item in a dependency forest is simply the total product of the root item's stochastic value and the arc costs along the path from the root to the item.

This dependency forest can be compacted into a dependency graph in which no item occurs in more than one node. That is, each graph node represents a stochastic item which is a coalesce of all the nodes in the dependency forest representing that particular item. The stochastic value of such an item is thus the sum of the posterior probabilities of the underlying items.

Figure 4.11 depicts the graphical relations of the items in the example state of GRA2 in Figure 4.9. We shall not attempt to depict the massively cyclic dependency graph of the start state for GRA3 (Figure 4.10) here.

4.5.3 Generating Linear Equations

Rather than attempting to write down a closed expression for the stochastic value of each item, we resort to creating a system of linear equations in terms of the stochastic values which encapsulate the possibly cyclic dependency structure of the items in the set.

Consider a state Ψ with k items, m of which are kernel items. That is, Ψ is the set of items $\{I_j \mid 1 \leq j \leq k\}$ such that I_j is a kernel item if $1 \leq j \leq m$. Again, let S_j be a variable representing the stochastic value of item I_j. The values of $S_1, \ldots, S_m$ are known since they can be computed as outlined in Section 4.5.1.

Consider a non-kernel item I_j, $m < j \leq k$. Let $\{I_{j_1}, \ldots, I_{j_{n'}}\}$ be the set of items in Ψ from which there is an arc into I_j in the dependency graph for Ψ. Also, let P_{j_i} denote the arc cost of the arc from item I_{j_i} to I_j. Then, the equation for the stochastic value of I_j, namely S_j, would be:

$$S_j = \sum_{i=1}^{n'} P_{j_i} \times S_{j_i} \tag{4.1}$$

Note that Equation (4.1) is a linear equation of at most $(k-m)$ unknowns, namely $S_{m+1}, \ldots, S_k$. This means that from 4.1 we have a system of $(k-m)$ linear equations with $(k-m)$ unknowns. This can be solved using standard algorithms like simple Gaussian Elimination [Strang, 1980].

The task of generating the equations can be further simplified by the following observations:

1. The cost of any incoming arc of a non-kernel item $I_i = [A_i \rightarrow \cdot\alpha_i, S_i]$ is the production probability of the production $\langle A_i \rightarrow \alpha_i, P_r \rangle$. In other words, $P_{j_i} = P_r$ for $i = 1 \ldots n'$. Equation (4.1) can then be simplified to $S_j = P_r \times \sum_{i=1}^{n'} S_{j_i}$.

2. Within a state, the non-kernel items representing any X-production have the same set of items with arcs into them. Therefore, these non-kernel items have the same value for $\sum_{x=1}^{n'} S_{j_x}$ (which is similar to the S_X in Section 4.5.1).

Thus, Equation (4.1) can be further simplified as $S_j = P_r \times S_{A_j}$ where $S_{A_j} = \sum_{x=1}^{n'} S_{j_x}$. With that, the system of linear equations for each state can be generated efficiently without having to construct explicitly the item dependency graph.

Examples

The system of linear equations for the state depicted in Figures 4.9 and 4.11 for grammar GRA2 is as follows: $S_0 = \frac{3}{7}$ *(Given)* $S_2 = \frac{2}{5}(S_0 + S_3)$; $S_1 = \frac{1}{2}(S_0 + S_3)$ $S_3 = \frac{1}{10}(S_0 + S_3)$

On solving the equations, we have $S_1 = \frac{5}{21}$, $S_2 = \frac{4}{21}$ and $S_3 = \frac{1}{21}$, which is the same solution as the one obtained by enumeration (Section 4.4.1).

Similarly, the following system of linear equations is obtained for the start state of massively left recursive grammar GRA3:

$S_0 = 1$ $S_6 = \frac{1}{3}(S_2 + S_5 + S_8)$
$S_1 = \frac{1}{7}(S_0 + S_1 + S_4 + S_7)$ $S_7 = \frac{1}{5}(S_3 + S_6 + S_9)$
$S_2 = \frac{4}{7}(S_0 + S_1 + S_4 + S_7)$ $S_8 = \frac{4}{15}(S_3 + S_6 + S_9)$
$S_3 = \frac{2}{7}(S_0 + S_1 + S_4 + S_7)$ $S_9 = \frac{1}{15}(S_3 + S_6 + S_9)$
$S_4 = \frac{1}{3}(S_2 + S_5 + S_8)$ $S_{10} = \frac{1}{3}(S_3 + S_6 + S_9)$
$S_5 = \frac{1}{6}(S_2 + S_5 + S_8)$ $S_{11} = \frac{2}{15}(S_3 + S_6 + S_9)$

On solving the equations, we have the solutions 1, $\frac{29}{77}$, $\frac{116}{77}$, $\frac{58}{77}$, $\frac{64}{77}$, $\frac{32}{77}$, $\frac{96}{77}$, $\frac{3}{7}$, $\frac{4}{7}$, $\frac{1}{7}$, $\frac{5}{7}$ and $\frac{2}{7}$ for the stochastic variables S_0 through S_{11} respectively.

4.5.4 Solving Linear Equations with Gaussian Elimination

The systems of linear equations generated during table construction can be solved using the popular method *Gaussian Elimination* which can be found in many numerical analysis or linear algebra textbooks (for example, Strang [Strang, 1980] or linear programming books (such as Vašek Chvátal, [Chvátal, 1983]). The basic idea is to eliminate the variables one by one by repeated substitutions. For instance, if we have the following set of equations:

(1) $S_1 = a_{11}S_1 + a_{12}S_2 + \ldots + a_{1n}S_n$

$\vdots$

(n) $S_n = a_{n1}S_1 + a_{n2}S_2 + \ldots + a_{nn}S_n$

We can eliminate S_1 and remove equation (1) from the system by substituting, for all occurrences of S_1 in equations (2) through (n), the right hand side of equation (1). We repeatedly remove variables S_1 through S_{n-1} in the same way, until we are left with only one equation with one variable S_n. Having thus obtained the value for S_n, we perform back substitutions until solutions for S_1 through S_n are obtained.

Complexity-wise, Gaussian elimination is a cubic algorithm [Chvátal, 1983] in terms of the number of variables (ie, the number of items in the closure set). The generation of linear equations per state is also polynomial since we only need to find the stochastic sum expressions — the S_{A_i}'s, for the nonterminals (Point 2 of Section 4.5.3). These expressions can be obtained by partitioning the items in the state set according to their left hand sides. There are $O(mn)$ possible LR(0) items (hence the size of each state is $O(mn)$) and $O(2^{mn})$ possible sets where n is the number of productions and m the length of the longest right hand side. Hence, asymptotically, the computation of the stochastic values would not affect the complexity of the algorithm, since it has only added an extra polynomial amount of work for each of the exponentially many possible sets.

Of course, we could have used other methods for solviug these linear equations, for example, by finding the inverse of the matrix representing the equations [Chvátal, 1983]. It is also plausible that particular characteristics of the equations generated by the construction algorithm can be exploited to derive the equations' solution more efficiently. We shall not discuss further here.

4.5.5 Stochastic Factors

Since the stochastic values of the terminal items in a parse state are basically posterior probabilities of that item given the root (kernel) item, the computation of the stochastic factors for the parsing actions, which is as presented in Wright and Wrigley [Wright and Wrigley, 1989], is fairly straightforward. For *shift*-action, say from State i to State $i+1$ on seeing the input symbol x, the corresponding stochastic factor for this action would be S_x, the sum of the stochastic values of all the leaf items in State i which are expecting the symbol x. For *reduce*-action, the stochastic factor is simply the stochastic value S_i of the item representing the reduction, namely $[A_i \rightarrow \alpha_i \cdot, S_i]$ if the reduction is via production $A_i \rightarrow \alpha_i$. For *accept*-action, the stochastic factor is the stochastic value S_n of the item $[S' \rightarrow S \cdot, S_n]$, since acceptance can be treated as a final reduction of the augmented production $S' \rightarrow S$, where S' is the system-introduced start symbol for the grammar.

4.6 Deferred Probabilities

The introduction of probability created a new criterion for equality between two sets of items: not only must they contain the same items, they must have the same item probability assignment. It is thus possible that we have many (possibly infinite) sets of similar items of differing probability assignments. This is especially so when there are loops amongst the sets of items (ie, the *states*) in the automaton created by the table construction algorithm — there is no guarantee that the differing probability assignments of the recurring states would converge. Even if they do converge eventually, it is still undesirable to have a huge parsing table of which many states have exactly the same underlying item set but differing probabilities.

To remedy this undesirable situation, we introduce a mechanism called *deferred probability* which will guarantee that the item sets converge without duplicating too many of the states. Thus far, we have been precomputing item's stochastic values in an *eager* fashion — propagating the probabilities as early as possible. Deferred probability provides a means to defer propagating certain problematic probability assignments (problematic in the sense that it causes many similar states with differing probability assignments) until appropriate. In the extreme case, probabilities are deferred until *reduction* time, ie, the stochastic factors of REDUCE actions are the respective rule probabilities and all other parse actions have unit stochastic factors. A reasonable postponement, however, would be to defer propagating the probabilities of the kernel items (kernel probabilities) until the following state. By forcing the differing item sets to have some fixed predefined probability assignment (while deferring the propagation of the "real" probabilities until appropriate times), we can prevent excessive duplication of similar states with same items but different probabilities.

To allow for deferred probabilities, we extend the original notion of probabilistic item to contain an additional field q which is the deferred probability for that item. That is, a probabilistic item would have the form $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q \rangle$. The default value of q is 1, meaning that no probability has been deferred. If in the process of constructing the closure states the table-construction program discovers that it is re-creating many states with the same underlying items but with differing probabilities or when it detects a non-converging loop, it might decide to replace that state with one in which the original kernel probabilities are deferred. That is, if the item $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q \rangle$ is a kernel item, and $\beta \neq \epsilon$, we replace it with a deferred item $\langle A \rightarrow \alpha \cdot \beta,\ p',\ \frac{pq}{p'} \rangle$ and proceed to compute the closure of the kernel set as before (ie, ignoring the deferred probabilities). In essence we have reassigned a kernel probability of p' to the kernel items *temporarily* instead of its original probability. It is important that this choice of assignment of p' be fixed with respect to that state. For instance, one assignment would be to impose a uniform probability distribution onto the deferred kernel items, that is, let p' be the probability $\frac{1}{\textit{Number of kernel items}}$. Another choice is to assign unit probability to each of the kernel items, which allows us to simulate the effect of treating each of the kernel items as if it forms a separate state.

Although in theory it is possible to defer the kernel probabilities until reduction time, in practice it is sufficient to defer it for only one state transition. That is, we recover the deferred probabilities in the next state. We can do this by enabling the propagation of the deferred probabilities in the next state, simply by multiplying back the deferred probabilities q into the kernel probabilities of the next state. In other words, as in Section 4.5.1, if $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i, q]$ is in State $m-1$, then the corresponding kernel item in State m would be $[A_i \rightarrow \alpha_i X \cdot \beta_i, \frac{S_i q}{S_X}, 1]$.

4.7 Algorithm Specification

A full algorithm for probabilistic LR parsing table construction for general probabilistic context-free grammar is presented here. The deferred proba-

bility mechanism as described in Section 4.6 is employed, the chosen reassignment of kernel probability being the unit probability.

4.7.1 Auxiliary Functions

CLOSURE

CLOSURE takes a set of ordinary nonprobabilistic LR(0) items and returns the set of LR(0) items which is the closure of the input items. A standard algorithm for CLOSURE can be found in [Aho and Ullman, 1977].

PROB-CLOSURE

Input: A set of k probabilistic items for some $k \geq 1$: $\{[A_i \rightarrow \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \leq i \leq k\}$.

Output: A set of probabilistic items which is the closure of the input probabilistic items. Each probabilistic item in the output set carries a stochastic value which is the sum of the posterior probabilities of that item given the input items.

Method:

Step 1: Let $\mathcal{C} :=$ CLOSURE($\{[A_i \rightarrow \alpha_i \cdot \beta_i] \mid 1 \leq i \leq k\}$);

Step 2: Suppose k' is the size of $\mathcal{C}$. Let I_i be the i-th item $[A_i \rightarrow \alpha_i.\beta_i]$ in $\mathcal{C}$, $1 \leq i \leq k'$. Also, for each item I_i, let S_i be a variable denoting its stochastic value.

1. For $1 \leq i \leq k$, $S_i := p_i$;
2. Let $\mathcal{E}_B$ be the set of items in $\mathcal{C}$ that are expecting B as the next symbol on the stack. That is, $\mathcal{E}_B$ is the set

$$\{I_j \mid I_j \in \mathcal{C},\ I_j = [A_j \rightarrow \alpha_j \cdot B\beta_j]\}$$

Let $S_B \stackrel{\text{def}}{=} \sum_{I_j \in \mathcal{E}_B} S_j$, where $B \in N$. For $k < i \leq k'$ such that $I_i = [A_i \rightarrow \cdot \beta i]$, set $S_i := P_r \times S_{A_i}$, where P_r is the probability of the production $A_i \rightarrow \beta_i$.

Step 3: Solve the system of linear equations generated by **Step 2**, using any standard algorithm such as simple Gaussian Elimination [Strang, 1980].

Step 4: Return $\{[A_i \rightarrow \alpha \cdot \beta, S_i, q_i] \mid 1 \leq i \leq k'\}$, where $q_i = 1$ for $k \leq i \leq k'$.

GOTO

Another useful function in table construction is GOTO($\{I_1 \ldots I_n\}, X$), where the first argument $\{I_1 \ldots I_n\}$ is a set of n probabilistic items and the second argument X a grammar symbol in $(N \cup T)$.

Suppose the probabilistic items in $\{I_1 \ldots I_n\}$ are such that those with symbol X after the dot are $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i, q_i]$, $1 \leq i \leq k$ for some $1 \leq k \leq n$. Let S_X be $\sum_{i=1}^{k} S_i$ and set GOTO($\{I_i\}, X$) to be PROB-CLOSURE($\{[A_i \rightarrow \alpha_i X \cdot \beta_i, \frac{S_i q_i}{S_X}, 1] \mid 1 \leq i \leq k\}$).

When $k = 0$, GOTO($\{I_i\}, X$) is undefined.

Sets-of-Items Construction

Let $\mathcal{U}$ be the canonical collection of sets of probabilistic items for the grammar G'. $\mathcal{U}$ can be constructed as described below.

Initially $\mathcal{U} :=$ PROB-CLOSURE($\{[S' \rightarrow \cdot S, 1]\}$). Repeat the process of applying the GOTO function (as defined in Step 4.7.1) with the existing sets in $\mathcal{U}$ and symbols in $(N \cup T)$ to generate new sets to be added to $\mathcal{U}$. If it is detected that an excessive number of states with similar underlying item sets but differing probabilities are created, use a state that is created by deferring the probabilities of the kernel items. That is, suppose the original kernel set is $\{[A_i \rightarrow \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \leq i \leq k\}$, use instead $\{[A_i \rightarrow \alpha_i \cdot \beta_i, 1, p_i q_i] \mid 1 \leq i \leq k \text{ and } \beta_i \neq \epsilon\}$.

The process stops when no new set can be generated.

Note that equality between two sets of probabilistic items here requires that they contain the same items with equal corresponding stochastic values, as well as deferred probabilities.

4.7.2 LR Table Construction

The algorithm is very similar to standard LR table construction [Aho and Ullman, 1977] except for the additional step to compute the stochastic factor for each action (*shift*, *reduce*, or *accept*).

Given a grammar $G = \langle N, T, R, S \rangle$, we define a corresponding grammar G' with a system-generated start symbol S':

$$\langle N \cup \{S'\}, T, R \cup \{< S' \rightarrow S, 1 >\}, S' \rangle.$$

Input: $\mathcal{U}$, the canonical collection of sets of probabilistic items for grammar G'.

Output: If possible, a probabilistic LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method: Let $\mathcal{U} = \{\Psi_0, \Psi_1, \ldots, \Psi_n\}$, where Ψ_0 is that initial set in Sets-of-Items Construction. The states of the parser are then $0, 1, \ldots, n$, with state i being constructed from Ψ_i. The parsing actions for state i are determined as follows:

1. If $[A \rightarrow \alpha \cdot a\beta, q_a]$ is in Ψ_i, $a \in T$, and $\text{GOTO}(\Psi_i, a) = \Psi_j$, set ACTION$[i, a]$ to $\langle$ "*shift* j", $p_a\rangle$ where p_a is the sum of q_a's – that is the stochastic values of items in Ψ_i with symbol a after the dot.
2. If $[A \rightarrow \alpha\cdot, p]$ is in Ψ_i, set ACTION$[i, a]$ to $\langle$ "*reduce* $A \rightarrow \alpha$", $p\rangle$ for every $a \in$ FOLLOW(A).
3. If $[S' \rightarrow S\cdot, p]$ is in Ψ_i, set ACTION$[i, \$]$ ($\$$ is an end-of-input marker) to $\langle$ "*accept*", $p\rangle$.

The goto transitions for state i are constructed in the usual way:

4. If $\text{GOTO}(I_i, A) = I_j$, set GOTO$[i, A] = j$

All entries not defined by rules (1) through (4) are made "*error*".

The FOLLOW table can be constructed from G by a standard algorithm in [Aho and Ullman, 1977].

4.8 Sammary

In this chapter, we have presented a method for dealing with left recursions in constructing probabilistic LR parsing tables for left recursive PCFGs. We have described runtime probabilistic LR parsers which use probabilistic parsing table. The table construction method outlined in this chapter has been implemented in Common Lisp. The two versions of runtime parsers described in this chapter have also been implemented in Common Lisp, and incorporated with various search strategies such as beam-search and best-first search (only for the tree-stack version) for comparison.

Chapter 5

Parsing Word Lattices

5.1 Introduction

This chapter is concerned with the problem of parsing word lattices [1]. A word lattice is an efficient representation of a large set of possible sentence candidates, of which only a few are grammatical and thus parsable. Word lattices are a common output of some speech recognizers, and may also arise as a result of multiple part-of-speech tags of sentence words. In the speech case, individual word hypotheses are characterized by a time interval (marking the beginning and ending times of the word) and a likelihood score. In the case of multiple part-of-speech tags, the order of words of the original sentence determines an order on the assigned parts-of-speech, and each possible part-of-speech tagging of a word is assigned a probability. In both cases, the lattice consists of a two dimensional grid, on which all the hypotheses are represented according to their time and probability features.

Parsing a word lattice involves finding a path of time-wise connecting words within the lattice that is grammatical. The goal of the parser is to find the grammatical path of highest overall score within the lattice. We describe an efficient algorithm for parsing such word lattices. Our algorithm is based on a Generalized LR style substring parser, that can parse an input string in arbitrary order. An efficient computation strategy is achieved by using an A^* heuristic to determine the order in which words of the lattice are processed.

[1] Major parts of this chapter are based on previously published papers [Lavie and Tomita, 1993a, Tomita, 1986, Lavie and Tomita, 1993b]. I would like to acknowledge the coauthor of the papers, Alon Lavie, whose contribution is included in this chapter.

Without the grammaticality constraint, the highest scoring path of time-wise connecting words through the lattice can be computed in time linear in the number of words in the lattice using a Dynamic Programming Algorithm [Thompson, 1990, Thompson, 1989].

Previous work by Tomita, Kita, Saito and others [Tomita, 1986, Kita *et al.*, 1989a, Saito and Tomita, 1988a] has focussed on how to use the predictive power of the Generalized LR parser in order to guide the search through the huge space of word hypotheses in speech recognition systems, and has also tried to deal with the problem of missing words. Saito [Saito, 1990] suggested an algorithm that can start parsing from an identified anchor word, from which the parsing can proceed sequentially in both directions. These parsing methods are rigid due to the fact that the parser must scan and process the input in a sequential uni-directional fashion.

Chow and Roukos [Chow and Roukos, 1989] describe a bottom-up CYK-style parsing algorithm that does not suffer from the uni-directionality restriction. The algorithm uses Dynamic Programming and Chart Parsing techniques in order to parse the word lattice and find the highest scoring grammatical path.

The word lattice parsing algorithm we present in this chapter has several advantages over the Chow and Roukos algorithm. First, our algorithm is founded on Generalized LR parsing, which is very efficient in practice due to the utilization of parsing tables that are pre-compiled in advance from the grammar. Second, our algorithm is based on an algorithm for parsing substrings that we have developed. This substring parsing algorithm follows from previous work by Bates and Lavie [Bates and Lavie, 1992],[Bates and Lavie, 1991] on recognizing substrings of LR languages, and from work by Rekers and Koorn [Rekers and Koorn, 1991]. The substring parser is converted into a GLR parser that can parse the words of an input sentence in any arbitrary order. Words of the input are parsed as substrings and are combined with other neighboring substrings as these become available. A unique feature of our substring parsing algorithm is that it can parse arbitrary substrings, irrespective of phrase boundaries. This allows neighboring substrings to be combined even when they span several partial phrases. This property provides complete flexibility in determining the order in which to parse the words of the input.

In order to achieve an efficient computation strategy for parsing the word lattice, we develop an A^* style heuristic. The heuristic determines the order in which lattice words are parsed so that potentially more probable substrings are pursued first. The heuristic guarantees a halting condition, by

which it can be determined when the best full parse found so far is the most probable one in the lattice.

The remainder of the chapter is organized in the following way. Section 5.2 describes the substring parsing algorithm. Section 5.3 presents our GLR arbitrary word order parser. A running example of the parser is presented in section 5.4. In Section 5.5 we describe the A^* style heuristic and how it is incorporated with the parser in order to efficiently parse the word lattice.

5.2 The Substring Parsing Algorithm

Our substring parsing algorithm is similar in principle to the one described by Rekers and Koorn [Rekers and Koorn, 1991], and follows from an algorithm for recognizing substrings of LR languages developed by Bates and Lavie [Bates and Lavie, 1992, Bates and Lavie, 1991]. For simplicity, we assume the parser is SLR(1), although the principles described here are applicable to the other LR parsing variants as well. Given an input $x = x_1 x_2 \cdots x_n$, the algorithm first accesses the parsing table in search of all states that wish to shift the first input symbol x_1. These states are entered into the GSS as initial states. Parsing contiuues from all of these initial GSS states in the ordinary way specified by the GLR parsing algorithm. For each top state in the GSS, the next action (or actions) is determined from the parsing tables, according to the state and the next input symbol. Each action may be either a *shift*, an *error* or a *reduce*, and is treated in the following manner:

- A *shift* action of the form sh$k$ (shift to state k) is treated normally. The input symbol is shifted into the GSS, and a new top state node with state k is added to the GSS.
- An *error* action indicates that the input cannot continue to be parsed from this top uode, and this path in the GSS is discarded.
- A *reduce* action of the form ri (reduce by rule i) is treated normally, as long as the reduction can be completed with the existing stack symbols in the GSS. If this is not the case, the reduce action is a *long reduction*, and is handled in a special way, as shall be described below.

The major difference between our algorithm and that proposed by Rekers and Koorn is iu the handling of long reductions. Long reductions are reductions that attempt to pop states and symbols beyond the bottom of

the GSS. They thus correspond to reductions that include symbols that are prior to the beginning of the given input string. Our algorithm uses an additional parsing table, the *long reduction goto table*, to handle such cases. The idea behind the long reduction goto table is to determine the set of states in which the parser could find itself after completing the reduction (or perhaps multiple reductions), and from which the next action would be a shift. It therefore enables the parser to postpone the actual performance of the reduction, and to continue parsing the input by shifting the next input symbol. For each possible state k and rule i, the table specifies the state (or states) to which the parser would goto after completing a reduction of rule i from top state k. The long reduction goto table is easily constructible in advance from the grammar in a way similar to the other parsing tables.

When our substring parser encounters a long reduction, it marks the top state in which the long reduction occurred, determines the set of continuation states from the long reduction goto table and adds these states as new top states to the GSS, connecting the new states with the old marked state. This action is in fact equivalent to delaying the actual reduction from taking place, and allows the parser to continue parsing the input as if the reduction had occurred. It is compatible with the action performed by the Rekers and Koorn algorithm in this case, but intentionally does not remove the reduced nodes from the GSS.

If the algorithm succeeds to reach the end of the input string x and has processed the last input symbol x_n, it is guaranteed by properties of the LR parsing paradigm that x is a valid substring of some sentence in the language described by the grammar, and as such is accepted by the substring algorithm. The algorithm does not produce a full or partial parse tree of the substring. However, the parse information is represented in the parser's GSS when the substring algorithm terminates, and is utilized by the arbitrary word order full-string parsing algorithm, that is based on our substring parser and presented in the following section.

5.3 Arbitrary Word Order Parsing

We now describe our arbitrary word order parsing algorithm that is based on the substring parsing algorithm presented in the previous section. The primary advantage of this algorithm is that the input word sequence may be parsed in an arbitrary chosen order. Furthermore, this order need not be determined prior to the start of the parsing process, and decisions on which

word of the input to handle next can be made dynamically, based on any kind of relevant information or heuristic. This property enables us to use an A^* heuristic to efficiently parse word lattices.

The main idea behind the algorithm is to efficiently parse islands of the input as substrings. Thus, key features of the substring recognition algorithm described in the previous section are used. Parsed islands of the input must be correctly combined with neighboring islands as these become available. Eventually, the parsed islands combine to a fully connected substring parse of the input. Additional constraints may be applied at this point in order to guarantee that the algorithm accepts only full-strings of the language.[2]

5.3.1 Description of the Algorithm

Due to space limitations and for the sake of simplicity, we describe here only the recognition aspect of the algorithm. However, the manipulation of pointers to maintain and eventually produce the parse tree (or trees) of the input are similar in nature to the corresponding actions in the Generalized LR parsing algorithm. To simplify the description, we assume the input is an n word sequence, where the i-th word is time tagged by the interval $[i-1, i]$. Parsed islands are marked with the interval $[i, j]$ which they span.

The beginning of an island is with the startup of a substring parse of a single word $[i-1, i]$. As in the initial stage of the substring parsing algorithm, the parsing table is searched for states that wish to shift the input word. These states are entered into the GSS and the shift action is performed.

After the initial shift action, reductions are performed. Normal reductions occur as usual. Long reductions are handled in the way described by our substring algorithm. The top state in the GSS is marked, and the long reduction goto table is accessed to determine the continuation states. When the parsing of an island reaches the stage where no more reductions can be performed on its GSS (the actions specified from all of the top nodes of the GSS are all shift actions), the processing of the island is stalled until it can combined with a neighboring island.

When two neighboring islands are combined, their GSSs are "glued" together. The GSS of the left island serves as the "lower" part, while that of the right island is the "upper" part. Each top state in the lower GSS is matched with corresponding bottom states of the upper GSS and the

[2]However, the parse through the entire input may in fact be valuable even in cases where the input is merely a substring (but not a full-string) of a sentence in the language.

structures are then merged. Parts of either of the two GSSs that find no matching part in their counterpart GSS are discarded at this point.

After the two GSS structures are combined, the upper part of the GSS (the part that originally belonged to the right island) is re-scanned in search of nodes marked by long reductions. A reduction previously marked as long, that can now be performed due to the more complete merged GSS, is executed at this point. This operation may rule out some of the continuation states that were determined when the long reduction occurred. The GSS is pruned accordingly in such cases. The result of the merging of the two GSSs and the post-processing described above is a GSS appropriate to the newly constructed joint island.

Attempts to combine islands occur in both directions. An island first attempts to combine with a neighboring left island. If no left neighboring island is available, the island attempts to combine with a neighbor to its right. When neither a left or a right neighboring island is available, the processing of the current island is stalled. The island will be picked up again when the parsing of a neighbor to either its left or right attempts to combine with it.

The fact that island combinations are attempted from both left and right directions guarantees that the algorithm will not deadlock, as long as some progress can be made. Thus, if the entire sentence is indeed parsable, the algorithm will eventually combine all islands into a single parsed island.

We assume the parser is able to distinguish if the input segment being processed starts at the beginning of the input or reaches the end of the input. If the algorithm is to accept only full-strings, this information can be used to constrain the parsing process in the following way. If the island does not reach the end of input, parsing actions for all possible following words (other than the end-of-input symbol "$") are considered and pursued. However, if the island does reach the end of the input, only the actions indicated for the end-of-input symbol ("$") need be performed. Similarly, if the segment spanned by the island starts at the beginning of the input, reductions that would require symbols prior to the beginning of the input can be ignored.[3]

The algorithm terminates when no progress can be done on any of the existing islands. The input is accepted if there exists a single island at this point, and the GSS contains the single accepting state. If on the other hand

[3]If, on the other hand, we wish the algorithm to accept inputs that are merely substrings (but not full-strings) of a sentence in the language, actions for all possible input words must be pursued at all times.

(1) S → NP VP
(2) NP → n
(3) NP → NP PP
(4) VP → v NP
(5) PP → p NP

Figure 5.1: An Example Grammar

there exist two or more islands, none of which can be combined, the input in whole is not a valid substring and is rejected. The existing islands at this point correspond to the largest valid substrings that could be found within the input string.

5.4 An Example

To clarify how our proposed arbitrary word order parsing algorithm operates in practice, we now present an example. The grammar in Figure 5.1 is a simple natural language grammar. From this grammar, we construct the standard SLR(1) parsing table of Table 5.1. Note that the table contains a "Shift/Reduce" conflict for state number 9, in the case of a preposition (terminal symbol "p"). This is due to an ambiguity with prepositional phrase attachments. The long reduction goto table for this parsing table is presented in Table 5.2. Note that reductions are unique per state in this case, therefore the long reduction goto states are a function of state only (and not of state and rule).

We now follow the first few steps of the arbitrary word order parsing algorithm on the input:
x = n v n p n p n
Each word of the input is tagged with its appropriate interval of the form $[i-1, i]$ (for $0 \leq i \leq 7$ in our case). Let us assume that the order of word processing chosen is that in which we process the input from the last word to the first.

We thus begin with the island $(\mathrm{n}, [6, \$])$. The tree on the left of Figure 5.2 is the initial GSS constructed, after shifting the input symbol "n". A normal reduction by rule 2 is then performed, resulting in the GSS shown in the middle of the figure. Since this island borders the end of the input, only further actions on the end-of-input symbol "$" are pursued. The action on

	Action				Goto			
State	n	v	p	$	NP	VP	PP	S
1	sh2				3			4
2		r2	r2	r2				
3		sh6	sh5			7	8	
4				acc				
5	sh2				9			
6	sh2				10			
7				r1				
8		r3	r3	r3				
9		r5	r5,sh5	r5			8	
10			sh5	r4			8	

Table 5.1: Standard Parsing Table for Grammar in Figure 1

Top state	Goto states after reduction
1	
2	3 9 10
3	
4	
5	
6	
7	
8	3 9 10
9	3 9 10
10	

Table 5.2: Long reduction goto table for the parsing table in Table 1

state 3 indicates an error, therefore the left tree of this GSS is discarded, and we remain with the GSS on the right side of the figure. The other actions are reductions that are all long at this point. Thus, the two top state nodes are marked (in the figure marked nodes are indicated by a double circle). Since this island borders the end-of-input, it will not be combinable with anything to its right. Therefore, there is no need to determine continuation states in this case. Since no neighboring islands are available at this point, the processing of the island is stalled.

We next continue with the island (p,[5,6]). The initial shift action results in the GSS shown in Figure 5.3. The only action from state 5 is a shift. At this point the island is ready to be combined with neighboring islands. Since there is no left neighbor available, the island is combined with the neighbor to its right to form the island (p n,[5,$]). The GSS that results from the combination is shown on the left of Figure 5.4. The delayed reduction by rule 5 from state 9 can now be performed, and this results in the GSS shown on the right side of the figure. The reduction by rule 3 from state 8 cannot be done (it is a long reduction), so the node is marked. Once again, the long reduction goto table is not accessed in this case, since the island borders the end-of-input.

The processing now proceeds to the island next in line, which is (n,[4,5])). The tree on the left of Figure 5.5 represents the initial GSS constructed, after the shifting of the input symbol "n". A normal reduction by rule 2 is then performed, resulting in the GSS shown in the middle of the figure. Note that the top node of state 9 now has two conflicting actions that need to be pursued. To achieve this we graphically split this node into two separate nodes.[4] One of the actions is a reduction by rule 5. This is a long reduction, so the node is marked (graphically by a double circle), aud the long reduction goto table is accessed to determine the continuation states. The continuation states are 3, 9 and 10. Since there already exist top level nodes of all three of these states, the long reduction node is connected with these three nodes. The resulting GSS is displayed on the right of Figure 5.5. The island is now ready to be combined with its neighbors. Since there is no left neighbor available, the island combines with its right neighbor to form the island (n p n,[4,$]). The resulting GSS is shown in Figure 5.6. The delayed reduction from the top node with state 8 can partially be executed, and the resulting GSS is displayed on the right side of the figure. The action on the top node of state 3 indicates an error, and this part of the graph is

[4] In practice, the node does not have to be explicitly separated.

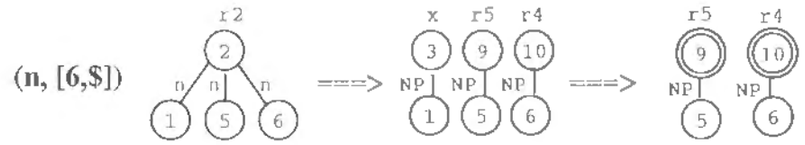


Figure 5.2: GSS of Island (n,[6,$])

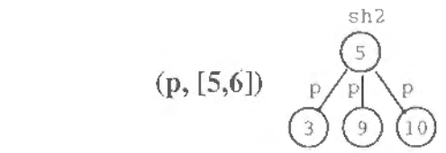


Figure 5.3: GSS of Island (p,[5,6])

thus deleted. The other reductions are long, but since the island borders the end of the input, no continuation states are added. the resulting GSS is shown at the bottom of Figure 5.6. Processing then moves on to the island (p,[3,4])) and continues from there on.

5.5 Using A^* Heuristic

We now turn to describe how to efficiently parse a word lattice, by determining an optimal ordering on the processing of the lattice words. The aim of the parser is to find a complete path of connecting lattice words that is parsable and is maximal in overall score. We assume that the overall score of a string of lattice words is simply the product of their individual scores. However, our analysis is just as valid with any other monotonically

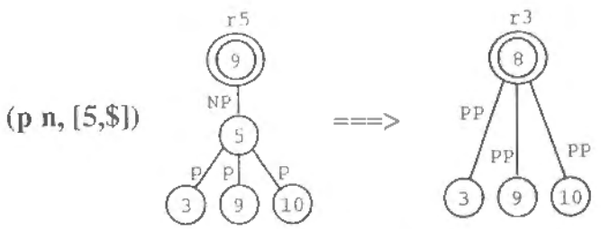


Figure 5.4: GSS of Island (p n,[5,$])

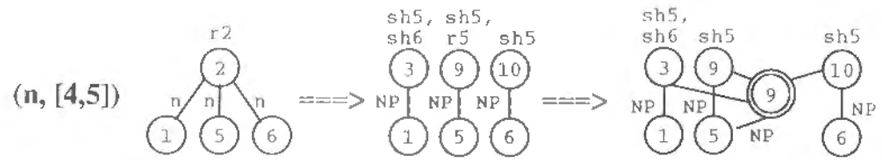


Figure 5.5: GSS of Island (n,[4,5])

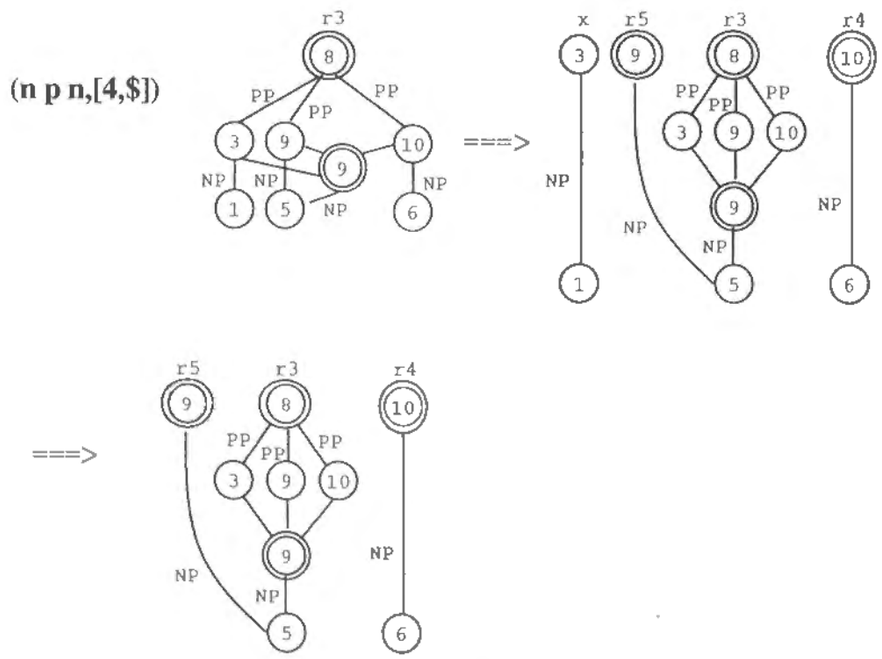


Figure 5.6: GSS of Island (n p n,[4,$])

increasing function of the individual scores.

5.5.1 The Heuristic

We suggest an A^* style heuristic to determine the ordering of the words. The lattice is then parsed using the parser described in the previous section, with the order of words determined by the heuristic.

The idea behind the heuristic is to attach an upper-bound score P^* to each lattice word and to each substring parsed by the algorithm. P^* is the maximal score of a path through the lattice that includes the word or substring. This path need not itself be grammatical. The P^* score of a word is defined in the following way:

- Let w be a given word of time span $[t_i, t_j]$ and score $P(w)$.
- Let P_0^{i*} be the score of the best path of connecting words from the beginning of the lattice ($time = 0$) to $time = t_i$.
- Let P_j^{e*} be the score of the best path of connecting words from $time = t_j$ to the end of the lattice ($time = end$).
- Then $P^*(w) = P_0^{i*} \cdot P(w) \cdot P_j^{e*}$

The P^* score of a parsed substring is defined similarly, where the product of P scores of the words of the substring is used instead of $P(w)$.

Note that any path that includes the word w and is parsable is guaranteed to have an overall score that is lesser or equal to $P^*(w)$. This condition holds for parsed substrings as well. Therefore, if the lattice words are processed by order of their P^* values, the following termination condition will hold:

- If the P^* score of all remaining unprocessed words of the lattice is less than or equal to the actual overall score of the best parsable full path found so far, than the algorithm can terminate.

The reason that this condition holds is that the current best parsable path is guaranteed to be better in score than any path that includes any of the words that haven't been yet processed, and thus the best parsable path has already been found.

5.5.2 Computing the P^* Scores

The P^* scores of words and substrings can be efficiently computed. In order to use the above mentioned equation, we must show how to compute, for each word w of time span $[t_i, t_j]$, the values P_0^{i*} and P_j^{e*}. For part-of-speech word lattices, this task is trivial, due to the simple time spans of the words. P_0^{i*} is simply the product of scores of the part-of-speech tags of greatest score that precede w. Similarly, P_j^{e*} is the prodnct of scores of the part-of-speech tags of greatest score that follow w.

In the case of speech produced word lattices, we can use the simple Dynamic Programming algorithm that finds the highest scoring (not necessarily grammatical) path of words through the lattice [Thompson, 1990], [Thompson, 1989]. Thompson's algorithm actually computes the desired values of P_0^{i*} as a by-product. By executing Thompson's algorithm time-wise in reverse (from the end of the lattice to its start), the values of P_j^{e*} are similarly computed as a by-product. The complexity of this algorithm is linear in the number of words in the lattice.

5.5.3 Parsing a Word Lattice using the Heuristic

Prior to the parsing itself, an initial phase must scan the lattice and assign to each word in the lattice its corresponding P^* score. Subsequently, the lattice words are sorted by their P^* scores. Words that have the same P^* score are ordered by their P score. The sorting allows the parsing algorithm to efficiently select the next word of the lattice that is to be parsed.

The parser processes the lattice word after word, in the order determined by the pre-sorting. The first word chosen to be processed is the word that has the greatest P score among the words of greatest P^* score. Each individually parsed word creates an initial island, which is then combined with all existing neighboring islands. Islands are stored in an list, which is also sorted according to the P^* scores of the islands.

Islands that correspond to full paths through the lattice, that are found to be valid full-parses, are stored in a third list, ranked by their actual overall score. Once the algorithm reaches the point where the P score of an existing full parse is greater or equal to the P^* score of the next word yet to be processed, it may terminate. The current best full parse is the desired solntion.[5]

[5] If *all* best solutions are desired, the algorithm must continue until the P score of the best full parse is strictly greater than the P^* score of the next word to be processed.

5.6 Summary

In this chapter we presented a new efficient algorithm for parsing word lattices. Based on an algorithm for parsing substrings, we developed a Generalized LR style parser that can parse an input string in any given word order. This algorithm parses words of the input as substrings, and combines these parsed islands with other neighboring islands as they become available. We described an A^* heuristic that can be used to impose an ordering on the lattice words. This heuristic guarantees an efficient computation strategy for finding the most probable grammatical path of words through the lattice.

The algorithm presented in this chapter has been implemented and tested on a small set of preliminary examples. However, we have yet to conduct a large scale experiment to evaluate the performance of the proposed algorithm in parsing actual speech produced word lattices. The efficiency of the parser with realistic large scale grammars will need to be tested as well.

Chapter 6

Noise Skipping Parsing

6.1 Introduction

In this chapter, we introduce a technique for substantially increasing the robustness of syntactic parsers to two particular types of extra- grammaticality: noise in the input, and limited grammar coverage [1]. Both phenomena cause a common situation, where the input contains words or fragments that are unparsable. The distinction between these two types of extra- grammaticality is based to a large extent upon whether or not the unparsable fragment, in its context, can be considered grammatical by a linguistic judgment. This distinction may indeed be vague at times, and practically unimportant.

Our approach to the problem is to enable the parser to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and focusing on the maximal subset of the input that is covered by the grammar. Although presented and implemented as an enhancement to the Generalized LR parsing paradigm, our technique is applicable in general to most phrase- structured based parsing formalisms. However, the efficiency of our parser is due in part to several particular properties of GLR parsing, and may thus not be easily transferred to other syntactic parsing formalisms.

The problem can be formalized in the following way: Given a context-free grammar G and a sentence S, find and parse S' - the largest subset of words of S, such that $S' \in L(G)$.

A naive approach to this problem is to exhaustively list and attempt to

[1] Substantial parts of this chapter are based on previously published paper [Lavie and Tomita, 1993c]. I would like to acknowledge the coauthor of the paper, Alon Lavie, whose contribution is included in this chapter.

parse all possible subsets of the input string. The largest subset can then be selected from among the subsets that are found to be parsable. This algorithm is clearly computationally infeasible, since the number of subsets is exponential in the length of the input string. We thus devise an efficient method for accomplishing the same task, and pair it with an efficient search approximation heuristic that maintains runtime feasibility.

The algorithm described in this paper, which we have named GLR*, is a modification of the Generalized LR parsing algorithm. It has been implemented and integrated with the latest version of the GLR Parser/Compiler [Tomita, 1990b, Tomita and Carbonell, 1987a].

There have been several other approaches to the problems of robust parsing, most of which have been special purpose algorithms. Some of these approaches have abandoned syntax as a major tool in handling extra-grammaticalities and have focused on domain dependent semantic methods [Carbonell and Hayes, 1984, Ward, 1991]. Other systems have constructed grammar and domain dependent fall-back components to handle extra-grammatical input that causes the main parser to fail [Stallard and Bobrow, 1992, Seneff, 1992].

Our approach can be viewed as an attempt to extract from the input the maximal syntactic structure that is possible, within a purely syntactic and domain independent setting. Because the GLR* parsing algorithm is an enhancement to the standard GLR context-free parsing algorithm, all of the techniques and grammars developed for the standard parser can be applied as they are. In particular, the standard LR parsing tables are compiled in advance from the grammar and used "as is" by the parser in runtime. The GLR* parser inherits the benefits of the original parser in terms of ease of grammar development, and, to a large extent, efficiency properties of the parser itself. In the case that the input sentence is by itself grammatical, GLR* behaves exactly as the standard GLR parser.

The remaining chapters of the paper are organized in the following way: Section 6.2 presents an outline of the basic GLR* algorithm itself, followed by a detailed example of the operation of the parser on a simple input string. In section 6.3 we discuss the search heuristic that is added to the basic GLR* algorithm, in order to ensure its runtime feasibility. We discuss an application of the GLR* algorithm to spontaneous speech understanding, and present some preliminary test results in section 6.4.

(1) S → NP VP
(2) NP → det n
(3) NP → n
(4) NP → NP PP
(5) VP → v NP
(6) PP → p NP

Figure 6.1: A Simple Natural Language Grammar

6.2 The GLR* Parsing Algorithm

The GLR* parsing algorithm is an extension of the Generalized LR Parser, described in chapter 2.

The parser accommodates skipping words of the input string by allowing shift operations to be performed from inactive state nodes in the Graph Structured Stack (GSS). Shifting an input symbol from an inactive state is equivalent to skipping the words of the input that were encountered after the parser reached the inactive state and prior to the current word being shifted. Since the parser is LR(0), reduce operations need not be repeated for skipped words (the reductions do not depend on any lookahead). Information about skipped words is maintained in the symbol nodes that represent parse subtrees.

An initial version of the GLR* parser has been implemented in Lucid Common Lisp, in the integrated environment of the Universal Parser Architecture.

6.2.1 An Example

To clarify how the proposed GLR* parser actually works, in lieu of a more formal description of the algorithm itself, we present a step by step runtime example. For the purpose of the example, we use a simple natural language grammar that is shown in Figure 6.1. The terminal symbols of the grammar are depicted in lower-case, while the non-terminals are in upper-case. The grammar is compiled into an SLR(0) parsing table, which is displayed in Table 6.1. Note that since the table is SLR(0), the reduce actions are independent of any lookahead. The actions on states 10 and 11 include both a shift and a reduce.

	Reduce	Shift					Goto			
State		det	n	v	p	$	NP	VP	PP	S
0		sh3	sh4				2			1
1						acc				
2				sh7	sh8			5	6	
3			sh9							
4	r3									
5	r1									
6	r4									
7		sh3	sh4				10			
8		sh3	sh4				11			
9	r2									
10	r5				sh8				6	
11	r6				sh8				6	

Table 6.1: SLR(0) Parsing Table for Grammar in Figure 1

To understand the operation of the parser, we now follow some steps of the GLR* parsing algorithm on the input `x = det n v n det p n`. This input is ungrammatical due to the second "`det`" token. The maximal parsable subset of the input in this case is the string that includes all words other than the above mentioned "`det`".

In the figures ahead, which graphically display the GSS of the parser in varions stages of the parsing process, we use the following notation:

- An *active* (top level) state node is represented by the symbol "`@`", with the state number indicated above it. Actions that are attached to the node are indicated to the right of the node.
- An *inactive* state node is represented by the symbol "`*`". The state number is indicated above the node and actions that are attached to the node are indicated above the state number.
- Grammar symbol nodes are represented by the symbol "`#`", with the grammar symbol itself displayed above it.

The parser operates in phases of shifts and reductions. We follow the GSS of the parser following each of these phases, while processing the input string. Reduce actions are distributed to the active nodes dnring initialization and after each shift phase. Shift actions are distributed after each reduce phase.

```
0                                   after initialization
@ sh3                               (and empty reduce phase)
```

Figure 6.2: Initial GSS

```
sh4                                 after first shift phase
 0  det  3                         (and empty reduce phase)
 *---#---@ sh9
```

Figure 6.3: GSS after first shift phase

Note that the GLR* parsing algorithm distributes shift actions to *all* state nodes (both active and inactive), whereas the original parser distributed shift actions only to active nodes. Reduce actions are distributed only to active state nodes.

Figure 6.2 is the initial GSS, with an active state node of state 0. Since there are no reduce actions from state 0, the first reduce phase is empty. With the first input token being "det", the shift action attached to state node 0 is "sh3".

Figure 6.3 shows the GSS after the first shift phase. The symbol node labeled "det" has been shifted and connected to the initial state node and to the new active state node of state 3. Since there are no reduce actions from state 3, the next reduce phase is empty. The next input token is "n". Shift actions are distributed by the algorithm to both the active node of state 3 and the inactive node of state 0, as can be seen in Figure 6.3.

Figure 6.4 shows the GSS after the next shift phase. The input token "n" was shifted from both state nodes, creating active state nodes of states 9 and 4. The shifting of the input token "n" from state 0 corresponds to a parsing possibility in which the first input token "det" is skipped. Reduce actions are distributed to both of the active nodes.

The following reduce phase reduces both branches into noun phrases. The two "NP"s are packed together by a local ambiguity packing procedure. Using information on skipped words that is maintained within the symbol


```
0  det  3   n   9                   after second shift phase
*---#---*---#---@ r2
|     n        4
|-----#--------@ r3
```

Figure 6.4: GSS after second shift phase

```
0  det  3   n   9                   after third reduce phase
*---#---*---#---*
|    n        4
|----#--------*
|        NP       2
|-------#-------@ sh7
```

Figure 6.5: GSS after third reduce phase

nodes, the ambiguity packing can detect that one of the noun phrases (the one that was reduced from "det n") is more complete, and the other noun phrase is discarded. The resulting GSS is displayed in Figure 6.5. Shift actions with the next input token "v" are then distributed to all the state nodes. However, in this case, only state 2 allows a shift of "v" into state 7.

Figure 6.6 shows the GSS after the third shift phase. The state 7 node is the only active node at this point. Since no reduce actions are specified for this state, the fourth reduce phase is empty. Shift actions with the next input token "n" are distributed to all state nodes, as can be seen in the figure.

Figure 6.7 shows the GSS after the fourth shift phase and Figure 6.8 after the fifth reduce phase. Note that there are no active state nodes after the fifth reduce phase. This is due to the fact that none of the state nodes produced by the reduce phase allow the shifting of the next input token "det". The original parser would have thus failed as this point. However, the GLR* parser succeeds in distributing shift actions to two inactive state nodes in this case.

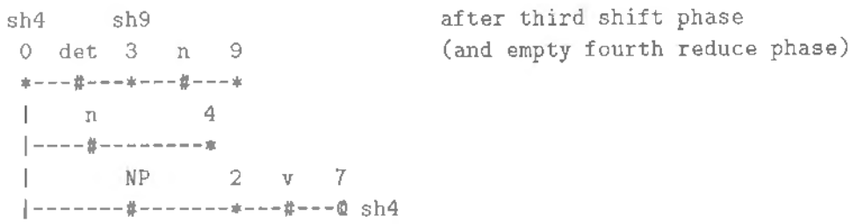


Figure 6.6: GSS after third shift phase

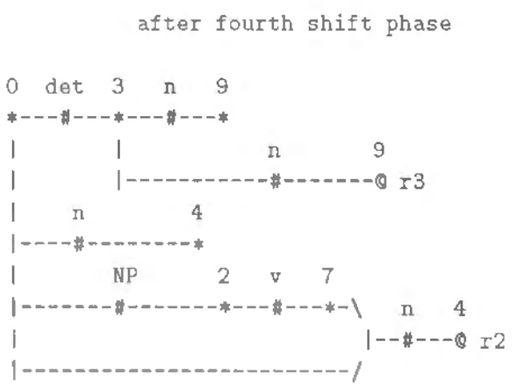


Figure 6.7: GSS after fourth shift phase

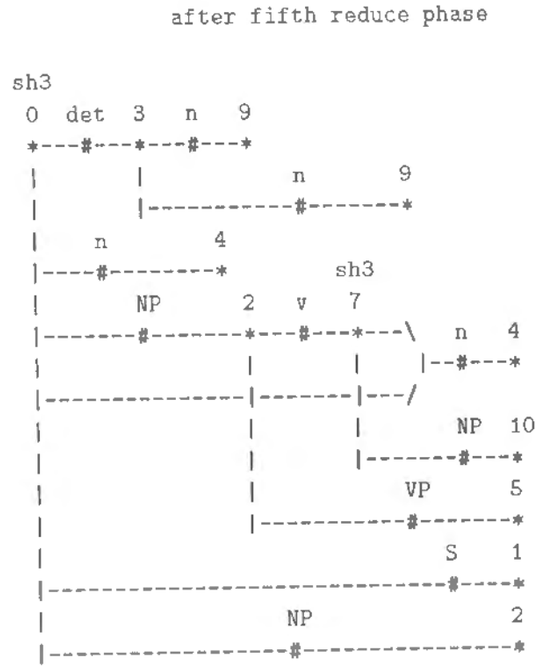


Figure 6.8: GSS after fifth reduce phase

For the sake of brevity we do not continue to further follow the parsing step by step. The final GSS is displayed in Figure 6.9. Several different parses, with different subsets of skipped words are actually packed into the single "S" node seen at the bottom of the figure. The parse that corresponds to the maximal subset of the input is the one in which the second "det" is the only word skipped.

6.2.2 Efficiency of the Parser

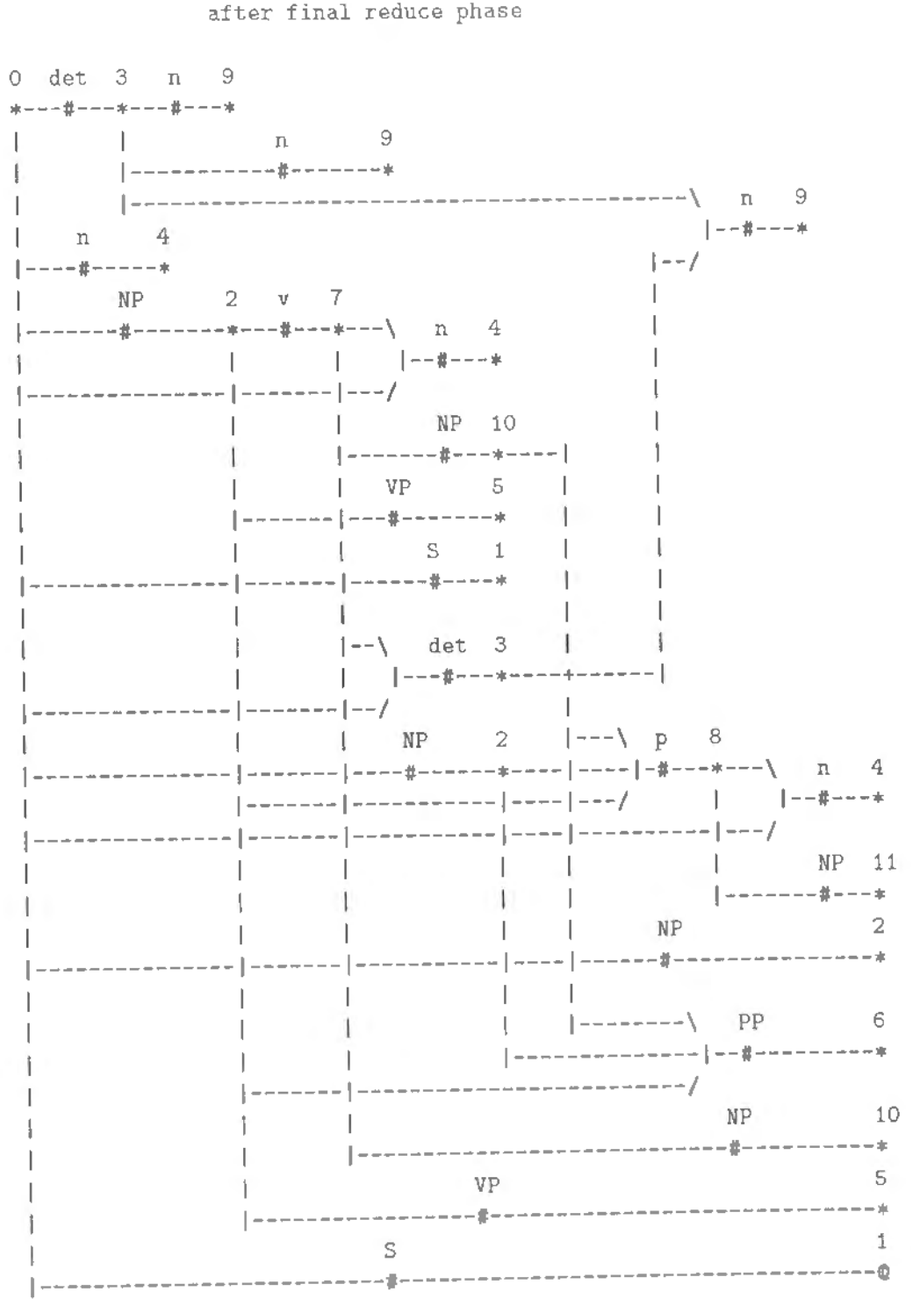
Efficiency of the parser is achieved by a number of different techniques. The most important of these is a sophisticated process of local ambiguity packing and pruning. A local ambiguity is a part of the input sentence that corresponds to a phrase (thus, reducible to some non-terminal symbol of the grammar), and is parsable in more than one way. The process of skipping words creates a large number of local ambiguities. For example, the grammar in Figure 6.1 allows both determined and undetermined noun phrases (rules 2 and 3). As seen in the example presented earlier, this results in the creation of two different noun phrase symbol nodes for the initial fragment "det n". The first node is created for the full phrase after a reduction according to the first rule. A second symbol node is created when the determiner is skipped and a reduction by the second rule takes place.

Locally ambiguous symbol nodes are detected as nodes that are surrounded by common state nodes in the GSS. The original GLR parser detects such local ambiguities and packs them into a single symbol node. This procedure was extended in the GLR* parser. Locally ambiguous symbol nodes are compared in terms of the words skipped within them. In cases such as the example described above, where one phrase has more skipped words than the other, the phrase with more skipped words is discarded in favor of the more complete parsed phrase. This subsuming operation drastically reduces the number of parses being pursued by the parser.

Another technique employed to increase the efficiency of the parser is the merging of state nodes of the same state after a reduce phase and after a shift phase. This allows the parsing through the GSS to continue with fewer state nodes.

6.2.3 Selecting the Best Maximal Parse

An obvious and unsurprising side effect of the GLR* parser is an explosion in the number of parses found by the parser. In principle, we are only in-



terested in finding the maximal parsable subset of the input string (and its parse). However, in many cases there are several distinct maximal parses, each consisting of a different subset of words of the original sentence. Additionally, there are cases where a parse that is not maximal in terms of the number of words skipped may be deemed preferable.

To select the "best" parse from the set of parses returned by the parser, we use a scoring procedure that ranks each of the parses found. We then select the parse that was ranked best. [2] Presently, our scoring procedure is rather simple. It takes into account the number of words skipped and the fragmentation of the parse (i.e. the number of S-nodes that the parsed input sentence was divided into). Both measures are weighed equally. Thus a parse that skipped one word but parsed the remaining input as a single sentence is preferred over a parse that fragments the input into three sentences, without skipping any input word.

On the top of our current research goals is the enhancement of this simple scoring mechanism. We plan on adding to our scoring function several additional heuristic measures that reflect various syntactic and semantic properties of the parse tree. We will measure the effectiveness of our enhanced scoring function in ranking the parse results by their desirability.

6.3 The Beam Search Heuristic

Although implemented efficiently, the basic GLR* parser is still not guaranteed to have a feasible running time. The basic GLR* algorithm described computes parses of all parsable subsets of the original input string, the number of which is potentially exponential in the length of the input string. Our goal is to find parses of maximal subsets of the input string (or almost maximal subsets). We have therefore developed and added to the parser a heuristic that prunes parsing options that are not likely to produce a maximal parse. This process has been traditionally called "beam search".

A direct way of adding a beam search to the parser would be to limit the number of active state nodes pursued by the parser at each stage, and continue processing only active nodes that are most promising in terms of the number of skipped words associated with them. However, the structure of the GSS makes it difficult to associate information on skipped words directly

[2] The system will display the n best parses found, where the parameter n is controlled by the user at runtime. By default, we set n to one, and the highest ranking parse is displayed.

with the state nodes. [3] We have therefore opted to implement a somewhat different heuristic that has a similar effect.

Since the skipping of words is the result of performing shift operations from inactive state nodes of the GSS, our heuristic limits the number of inactive state nodes from which a input symbol is shifted. At each shift stage, shift actions are first distributed to the active state nodes of the GSS. This corresponds to no additional skipped words at this stage. If the number of state nodes that allow a shift operation at this point is less than a predetermined constant threshold (the "beam-limit"), then shift operations from inactive state nodes are also considered. Inactive states are processed in an ordered fashion, so that shifting from a more recent state node that will result in fewer skipped words is considered first. Shift operations are distributed to inactive state nodes in this way until the number of shifts distributed reaches the threshold.

This beam search heuristic reduces the runtime of the GLR* parser to within a constant factor of the original GLR parser. Although it is not guaranteed to find the desired maximal parsable subset of the input string, our preliminary tests have shown that it works well in practice.

The threshold (beam-limit) itself is a parameter that can be dynamically set to any constant value at rnntime. Setting the beam-linit to a value of 0 disallows shifting from inactive states all together, which is equivalent to the original GLR parser. In preliminary experiments that we have conducted (see next section) we have achieved good resnlts with a setting of the beam-limit to values in the range of 5 to 10. There exists a direct tradeoff between the value of the beam-limit and the runtime of the GLR* parser. With a set value of 5, our tests have indicated a runtime that is within a factor of 2-3 times that of the original GLR parser, which amounts to a parse time of only several seconds on sentences that are up to 30 words long.

6.4 Parsing of Spontaneous Speech Using GLR*

6.4.1 The Problem of Parsing Spontaneous Speech

As a form of input, spontaneous speech is full of noise and irrelevances that surround the meaningful words of the utterance. Some types of noise can be detected and filtered out by speech recognizers that process the speech

[3] This is due to the fact that state nodes are merged, so that a state node may be common to several different parses, with different skipped words associated with each parse.

	Robust Parser
	number (and percent)
Parsable	99
Unparsable	1
Good/Close Parses	77
Bad Parses	22

Table 6.2: Performance of the GLR* Parser on Spontaneous Speech

signal. A parser that is designed to successfully process speech recognized input must however be robust to various forms of noise, and be able to weed out the meaningful words from the rest of the utterance.

When parsing spontaneous spoken input that was recognized by a speech recognition system, the parser must deal with three major types of extra-grammaticality:

- Noise due to the spontaneity of the speaker, such as repeated words, false beginnings, stutteriug, and filled pauses (i.e. "ah", "um", etc.).
- Ungrammaticality that is due to the language of the speaker, or to the coverage of the grammar.
- Noise due to errors of the speech recognizer.

We have conducted two preliminary experiments to evaluate the GLR* parser's ability to overcome the first two types of extra-grammaticality. We are in the process of experimenting with the GLR* parser on actual speech recognized output, in order to test its capabilities in handling errors produced by the speech recognizer.

6.4.2 Parsing of Noisy Spontaneous Speech

The first test we conducted was intended to evaluate the performance of the GLR* parser on noisy sentences typical of spontaneous speech. The parser was tested on a set of 100 sentences of transcribed spontaneous speech dialogues on a conference registration domain. The input is haud-coded transcribed text, not processed through any speech recognizer. The grammar used was an upgraded version of a grammar for the conference registration task, developed and used by the JANUS speech-to-speech translation project at CMU [Waibel et al. 1991]. Since the test sentences were drawn from

actual speech transcriptions, they were not guaranteed to be covered by the grammar. However, since the test was meant to focus on spontaneous noise, sentences that included verbs and nouns that were beyond the vocabulary of the system were avoided. Also pruned out of the test set were short opening and closing sentences (such as "hello" and "goodbye"). The transcriptions include a multitude of noise in the input. The following example is one of the sentences from this test set:

```
"fckn2_10 /ls/ /h#/ um okay {comma}
then yeah I am disappointed {comma}
*pause* but uh that is okay {period}"
```

The performance results are presented in Table 6.2. Note that due to the noise contaminating the input, the original parser is uuable to parse a single one of the sentences in this test set. The GLR* parser succeeded to return some parse result in all but one of the test seutences. However, since returning a parse result does not by itself guarantee an analysis that adequately reflects the meaning of the original utterance, we reviewed the parse results by hand, and classified them into the categories of "good/close" and "bad" parses. The results of this classification are included in the table.

6.4.3 Grammar Coverage

We conducted a second experiment aimed exclusively on evaluating the ability of the GLR* parser to overcome limited grammar coverage. In this experiment, we compared the results of the GLR* parser with those of the original GLR parser ou a common set of sentences using the same grammar. We used the grammar from the spontaneous speech experiment for this test as well. The common test set was a set of 117 sentences from the conference registration task of the JANUS project. These sentences are simple synthesized text sentences. They contain no spontaneous speech noise, and are not the result of any speech recognition processing. Once again, to evaluate the quality of the parse results returned by the parser, we classified the parse results of both parsers by hand into two categories: "good/close parses" and "bad parses". The results of the experiment are presented in Table 6.3.

The results indicate that using the GLR* parser results in a significant improvement in performance. The percentage of sentences, for which the parser returued good or close parses increased from 52% to 70%, an increase of 18%. Fully 97% of the test sentences (all but 3) are parsable by the GLR* parser, an increase of 36% over the original parser. However, this includes

	Original Parser		Robust Parser	
	number	percent	number	percent
Parsable	71	61%	114	97%
Unparsable	46	39%	3	3%
Good/Close Parses	61	52%	82	70%
Bad Parses	10	9%	32	27%

Table 6.3: Performance of the GLR* Parser vs. the Original Parser

a significant increase (from 9% to 27%) in the number of bad parses found. Thus, fully half of the additional parsable sentences of the set return with parses that may be deemed bad.

The results of the two experiments clearly point to the following problem: Compared with the GLR* parser, the original GLR parser, although fragile, returned results of relatively good quality, when it succeeded in parsing the input. The GLR* parser, on the other hand, will succeed in parsing almost any input, but this parse result may be of little or no value in a significant portion of cases. This indicates a strong need in the development of methods for discriminating between good and bad parse results. We intend to try and develop some effective heuristics to deal with this problem. The problem is also due in part to the ineffectiveness of the simple heuristics currently employed for selecting the best parse result from among the large set of parses returned by the parser. As mentioned earlier, we intend to concentrate efforts on developing more sophisticated and effective heuristics for selecting the best parse.

6.5 Summary

Motivated by the difficulties that standard syntactic parses have in dealing with extra-grammaticalities, we have developed GLR*, an enhanced version of the standard Generalized LR parser, that can effectively handle two particular problems that are typical of parsing spontaneous speech: noise contamination and limited grammar coverage.

Given a grammar G and and input string S, GLR* finds and parses S', the maximal subset of words of S, such that S' is in the language $L(G)$. The parsing algorithm accommodates the skipping of words and fragments of the

input string by allowing shift operations to be performed from inactive states of the GSS (as well as from the active states, as is done by the standard parser). The algorithm is coupled with a beam-search-like heuristic, that controls the process of shifting from inactive states to a limited beam, and maintains computational tractability.

Most other approaches to robust parsing have suffered to some extent from a lack of generality and from being domain dependent. Our approach, although limited to handling only certain types of extra-grammaticality, is general and domain independent. It attempts to maximize the robustness of the parser within a purely syntactic setting. Because the GLR* parsing algorithm is a modification of the standard GLR context-free parsing algorithm, all of the techniques and grammars developed for the standard parser can be applied as they are. In the case that the input sentence is by itself grammatical, GLR* behaves exactly as the standard GLR parser. The techniques used in the enhancement of the standard GLR parser into the robust GLR* parser are in principle applicable to other phrase-structure based parsers.

Preliminary experiments conducted on the effectiveness of the GLR* parser in handling noise contamination and limited grammar coverage have produced encouraging results. However, they have also pointed out a definite need to develop effective heuristics that can select the best parse result from a potentially large set of possibilities produced by the parser. Since the GLR* parser is likely to succeed in producing some parse in practically all cases, successful parsing by itself can no longer be an indicator to the value and quality of the parse result. Thus, additional heuristics need to be developed for evaluating the quality of the parse found.

Chapter 7

Speech Translation Systems

7.1 Introduction

In this chapter we first review speech recognition in terms of its historical background and current technology [1].

. Centrally, we address the issue of integrating speech and natural language analysis in general and in concrete systems. Unfortunately, the integration of speech recognition and language analysis is far from simple—direct end-to-end connection yields poor performance. Instead, both processes must be more tightly coupled with appropriate mutual feedback.

We then present three speech translation systems. Although all three are research projects at Carnegie Mellon University (CMU), they are representative of general approaches to the machine translation of speech. The systems are as follows:

- **SpeechTrans**, with noise-tolerant Generalized LR parsing;
- **Sphinx-LR**, with Hidden Markov Models-LR (HMM-LR); and
- **JANUS**, with linked predictive neural networks (LPNN).

[1] Substational parts of this chapter are based previously published papers [Tomita *et al.*, 1989, Nirenburg *et al.*, 1991, Tomita *et al.*, 1990a, Saito and Tomita, 1988b, Woszczyna *et al.*, 1993, ?, Tomita, 1988e, Tomabechi *et al.*, 1989, Tomita *et al.*, 1990b, ?]. I would like to acknowledge the authors of those papers whose contributions are included in this chapter: O. Barkai, Jaime Carbonell, Noah Coccaro, A. Eisele, Ken Goodman, T. Kawabata, Kenji Kita, Alon Lavie, Arthur McNair, Teruko Mitamura, See-Kiong Ng, Sergei Nirenburg, I. Rogina, Carolyn Rose, Hiroaki Saito, T. Sioboda, Hideto Tomabechi, Naomi Waibel, Alex Waibel, Wayne Ward, and Monica Woszczyna.

The systems are described in sections 7.3, 7.4 and 7.5, respectively.

7.2 Speech Recognition

Speech recognition is the process of mapping acoustic wave forms corresponding to spoken language into unique sequences of symbols, ideally strings of words. *Speech understanding* is sometimes used to mean only recognition and sometimes to encompass full language analysis.

Speech recognition is a very complex process that requires the discrimination of signals that vary in frequency, amplitude, onset time (phase) and temporal elasticity (speed of utterance) in different ways, by different speakers at different times. There have been many approaches to this task, but there seems to be no simple "magic bullet." Since progress is slow but steady and cumulative, different ways of characterizing system performance have evolved and gained widespread acceptance among researchers.

7.2.1 A Historical Perspective

Speech recognition has been studied fairly extensively for many years. Prototypes of the earliest successful large-scale systems reaching a 1,000-word vocabulary appeared about 1975, at the end of a five-year research plan by the U.S. government's Defense Advanced Research Projects Agency (DARPA). Two well-known examples are HEARSAY-II, which incorporated constraints, mostly syntactic, from language to facilitate recognition [Lesser *et al.*, 1975, Lea, 1980] and HWIM [Woods *et al.*, 1976, Wolf and Woods, 1980]. The HARPY system was of particular note, as it championed a different approach: It compiled rather than interpreted higher-level knowledge and used the beam-search technique to yield the best performance of the 1975 systems [Lowerre, 1976, Lowerre and Reddy, 1980]. Also in that year, Itakura [Itakura, 1975] of Nippon Telephone and Telegraph introduced the dynamic time warp (DTW) for nonlinear alignment of speech.

In 1982, Wilpon et al. at Bell Labs used clustering techniques to attempt speaker-independent isolated-word recognition. A recognition accuracy of 91% on a 129-word task was reported. The FEATURE system at CMU [Cole *et al.*, 1983] achieved an accuracy of greater than 90% in English letter recognition without grammar, using a feature-based approach.

In 1985, the IBM Speech Recognition Group addressed a natural very-large-vocabulary task and achieved impressive results. The Tangora system

obtained a 97% recognition rate for speaker-dependent recognition of sentences with clear pauses between words, using a 5,000-word vocabulary and a natural-language-like grammar with a perplexity of 160.

Bolt, Beranek and Newman's (BBN) BYBLOS system in 1987 used context-dependent modeling of phonemes and obtained a 93% accuracy on a 997-word continuous task [Chow *et al.*, 1987, Kubala *et al.*, 1988]. Using continuous HMM, a sentence recognitiou rate of 97.1% was achieved without the use of a grammar by Bell Labs on speaker-independent connected digit recognition [Rabiner *et al.*, 1988]. The Sphinx system at CMU, which uses Hidden Markov Modeling of speech, achieved speaker-independent word accuracies of 71%, 94% and 96% on the 997-word DARPA resource management task, with grammars of perplexity 997, 60, and 20, respectively, in 1988 [Lee, 1988].

There are several speech-to-speech MT projects under way throughout the world, primarily in Japan, Europe and the United States. For instance, in Japan, the ATR Interpreting Telephony Research Laboratories were established in 1986 to investigate automatic speech translation aids for overseas communications. Their research program, labeled "interpreting telephony," has the ambitious goal of enabling, in constrained domains, a person speaking one language to communicate readily by telephone with someone speaking another language. The integration of technologies in speech recognition, machine translation and speech synthesis is the focus of their investigations [Laboratories, 1989]. Another example is research at British Telecom Research Laboratories, which has been successful in overcoming some practical problems in the recognition, synthesis and translation of speech; their approach employs the use of carefully selected keywords [Stentiford and Steer, 1988]. The Center for Machine Translation at CMU is also deeply involved in speech-to-speech translation, and has produced several successful, prototype systems; some of these are described in the following sections.

7.3 SpeechTrans

The SpeechTrans project constitutes one of several efforts to integrate speech into a machine translation system at CMU's Center for Machine Translation. This project uses the Generalized LR parsing algorithm described in chapter 2 for language analysis with Matsushita experimental hardware for low-level phonemic speech recognition. It also used GENKIT version 3-2 (see Appendix B) for language generation with DEC Talk for speech synthesis.

Much effort has been devoted to make the SpeechTrans parser more robust against noise in order to analyze sentences that suffer from acoustical recognition errors. The speech recognition device, which is a high-speed, speaker-independent system developed by Matsushita Research Institute [Morii *et al.*, 1985] takes a continuously spoken Japanese utterance, for example *megaitai* ("I have a pain in my eye"), from speaker-microphone input and produces a sequence of phoneme symbols. Because the speech recognition device does not have any syntactic or semantic knowledge, recognition errors often produce illegal or *noisy* phoneme sequences, such as "ebaitaai" for *megaitai*.[2] Some more input/output examples of the speech device are presented in the following example, where the left-hand side of each line is the correct phoneme sequence and the right-hand side is what was recognized.

- *igamukamukasuru* → "igagukamukusjuru"
- *kubigakowabaqteiru* → "kubigakooboqteiiru"
- *atamagaitai* → "otomogaitai"

The task then is to parse noisy phoneme sequences like those in the above example and analyze the meaning of the original input utterances. A very efficient parsing method is crucial because the task's search space is much larger than that of parsing non-noisy sentences. In other words, one must attempt to parse variants of the input sequence to find the closest one that satisfies all lexical, syntactic and semantic well-formedness constraints. We adopt the Generalized LR parsing algorithm discribed in chapter 2, together with a scoring scheme to select the most likely sentence from multiple candidates. The use of a *confusion matrix*, created in advance by analyzing a large set of input/output pairs to improve the scoring accuracy, is discussed below.

Note that some speech recognition devices (such as Matsushita's) produce a phoneme sequence, not a phoneme lattice; there are no other phoneme candidates available as alternates. Therefore, at analysis time, we must make the best guess based solely on the phoneme sequence generated by the speech device. Errors caused by the speech device can be classified into three groups:

[2] We distinguish *noisy* from *ill-formed*. The former is due to recognition-device errors, while the latter is due to human users mistyping or misconstructing their sentences. Each phenomenon leads to different kinds of deviation from the correct or expected input.

- *Substituted phonemes:* phonemes recognized incorrectly. The second phon- eme /b/ in "ebaitaai" is a substituted phoneme, for example.
- *Deleted phonemes:* phonemes which are actually spoken but not recognized by the device. For example, a phoneme /m/ is deleted at the beginning of "ebaitaai."
- *Inserted phonemes:* phonemes recognized by the device but which are not actually spoken. The penultimate phoneme /a/ in "ebaitaai," for example, is an inserted phoneme.

To cope with these problems, we integrate two key technologies:

1. The GLR, a very efficient parsing algorithm, because our task requires much more search than conventional typed sentence parsing; and
2. A good scoring scheme, to select the most likely sentence from a multiple candidate set, as described in section 7.3.3.

SpeechTrans uses an augmented context-free grammar whose terminal symbols are phonemes rather than words. That is, the grammar contains rules like

```
Noun --> w a t a s i
```

instead of

```
Noun --> 'watasi'
```

Parsing therefore proceeds character by character (phoneme by phoneme). The grammar was developed for a doctor-patient communication task [Tomita and Carbonell, 1987b, Tomita *et al.*, 1988a, Tomita *et al.*, 1987] and consists of more than 2,000 rules including lexical rules like the one above.

7.3.1 Handling Erratic Phonemes

To cope with substituted, inserted and deleted phonemes, the parser must consider these errors as it parses an input from left to right. While the basic algorithm described in chapter 5 cannot handle these noisy phenomena, it is well suited to consider many possible parses at the same time. Therefore, it can be modified relatively easily to handle various noisy phenomena:

- *Substituted phonemes:* Each phoneme in a phoneme sequence may have been substituted and thus may be incorrect. The parser should consider all these possibilities. A phoneme lattice is created dynamically by placing alternate phoneme candidates in the same location as the original phoneme. Each possibility is then explored by each branch of the parser. Not all phonemes can be substituted for any other phoneme. For example, while /o/ can be misrecognized as /u/, /i/ can never be misrecognized as a consonant. This kind of information can be obtained from a *confusion matrix*, which we discuss in the next section. With the confusion matrix, the parser need not create an exhaustive set of alternate phoneme candidates, only the mutually confusible ones.
- *Inserted phonemes:* Each phoneme in a phoneme sequence may be an extra one, and the parser should consider the deletion of the current phoneme, assuming that at most one inserted spurious phoneme can exist between two actual phonemes.
- *Deleted phonemes:* Deleted phonemes can be handled by inserting potentially deleted phonemes between two actual phonemes. The parser assumes that at most one phoneme can be missing between two actual phonemes, and we have found the assumption quite reasonable. All possible legal insertions are considered if the current parse (without insertions) fails.

7.3.2 An Example

In this subsection, we present a sample trace of the parser. Here we use the grammar in figure 7.1 and the LR table in figure 7.2 to try to parse the phoneme sequence "ebaitaai." (The right sequence is "megaitai" which means "I have a pain in my eye.")

For this example we make the following assumptions for substituted and deleted phonemes:

- /i/ may be misrecognized as /e/
- /e/ may be misrecognized as /a/
- /g/ may be misrecognized as /b/
- /m/ has a high probability of being missed in the output sequence

```
-------------------------------
    (1)     S --> NP V
    (2)     S --> N
    (3)     S --> V
    (4)    NP --> N P
    (5)     N --> m e
    (6)     N --> i
    (7)     P --> g a
    (8)     V --> i t a i
-------------------------------
```

Figure 7.1: An Example Japanese Grammar

State	a	i	e	m	g	t	$	N	NP	P	V	S
0		s4		s5				2	3		1	6
1							r3					
2					s7,r2					8		
3		s9									10	
4					r6	s11,r6						
5			s12									
6							acc					
7	s13											
8		r4										
9						s11						
10							r1					
11	s14											
12					r5		r5					
13		r7										
14		s15										
15							r8					

Figure 7.2: LR Parsing Table for the Example Japanese Grammar

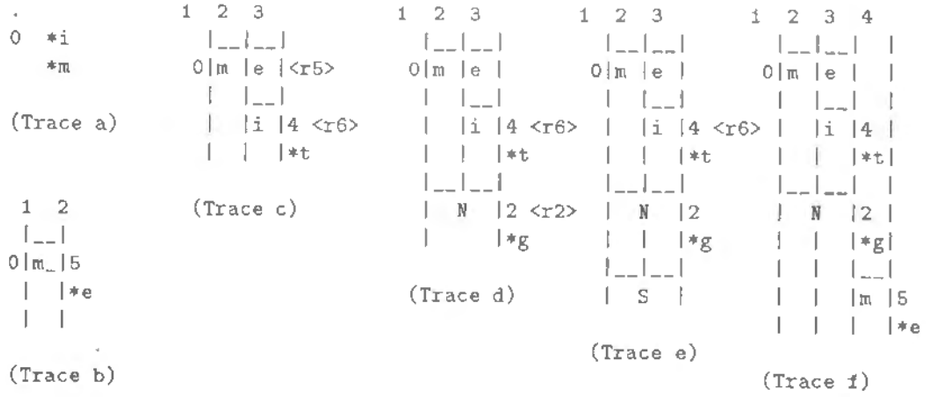


Figure 7.3: Example Trace a - f

First an initial state 0 is created. The action table indicates that the initial state is expecting "m" and "i" (figure 7.3a). Since the parsing strictly proceeds from left to right, the parser looks for the candidates of the missing phonemes between the first time frame 1 - 2. (We will use the term T1, T2, ... for representing the time 1, time 2, ... in figure 7.5.) Only the phoneme "m" in this group is applicable to state 0. The new state number 5 is determined from the action table (figure 7.3b).

The next group of phonemes between T2 and T3 consists of the "e" phoneme in the phoneme sequence and the altered candidate phonemes of "e". In this group "e" is expected by state 5 and "i" is expected by state 0(figure 7.3c). After "e" is taken, the new state is 12, which is ready for the action "reduce 5". Thus, using the rule 5(`N --> m e`), we reduce the phonemes "m e" into N. From state 0 with the nonterminal N, state 2 is determined from the goto table. The action table, then, indicates that state 2 has a multiple entry, i.e., state 2 is expecting "g" and ready for the reduce action(figure 7.3d). Thus, we reduce the nonterminal N into S by rule 2(`S --> N`), and the new state number 6 is determined from the goto table(figure 7.3e). The action table indicates that state 6 is an accept state, which means that "m e" is a successful parse. But only the first phoneme

"e" of the input sequence "ebaitaai" is consumed at this point. Thus we discard this parse by the following constraint.

> **Constraint 1:** The successful parse should consume the phonemes at least until the phoneme just before the end of the input sequence.

Note that only the parse S in figure 7.3e is ignored and that the nonterminal N in figure 7.3d is alive.

Now we return to the figure 7.3c and continue the shift action of "i". After "i" is taken, the new state 4 is determined from the action table. This state has a multiple entry, i.e. state 4 is expecting "t" and ready for the reduce action. Thus we reduce "i" into N by rule 6. Here we use the *local ambiguity packing* technique, because the reduced nonterminal is the same, the starting state is 0 for both, and the new state is 2 for both. Thus we do not create the new nonterminal N.

Now we go on to the next group of phonemes between T3 and T4. Only "m" is applied to the initial state (figure 7.4g).

The next group of phonemes between T4 and T5 has two applicable phonemes, i.e. "m" to state 0 and "g" to state 2. After "g" is taken, the new state 7 is determined from the action table (figure 7.4h).

The next group of phonemes between T5 and T6 has only one applicable phoneme; "m" to state 0. Here we can introduce another constraint which discards this partial-parse.

> **Constraint 2:** After consuming two phonemes of the input sequence, no phonemes can be applied to the initial state 0.

This constraint is natural because it is unlikely that more than two phonemes are recorded before the actual beginning phoneme for our speech recognition device.

The next group of phonemes between T6 and T7 has two applicable phonemes, i.e. "a" to state 7 and "e" to state 5. After "a" is taken, the new state 7 is ready for the reduce action. Thus, we reduce "g a" into P by rule 7(figure 7.4i). The new state 8 is determined by the goto table, and is also ready for the reduce action. Thus we reduce "N P" into NP by rule 4. The new state is 3. In applying "e", there are two "state 2"s: one is "m" between T1 and T2; the other one is "m" between T3 and T4. Here we can introduce a third constraint which discards the former partial-parse.

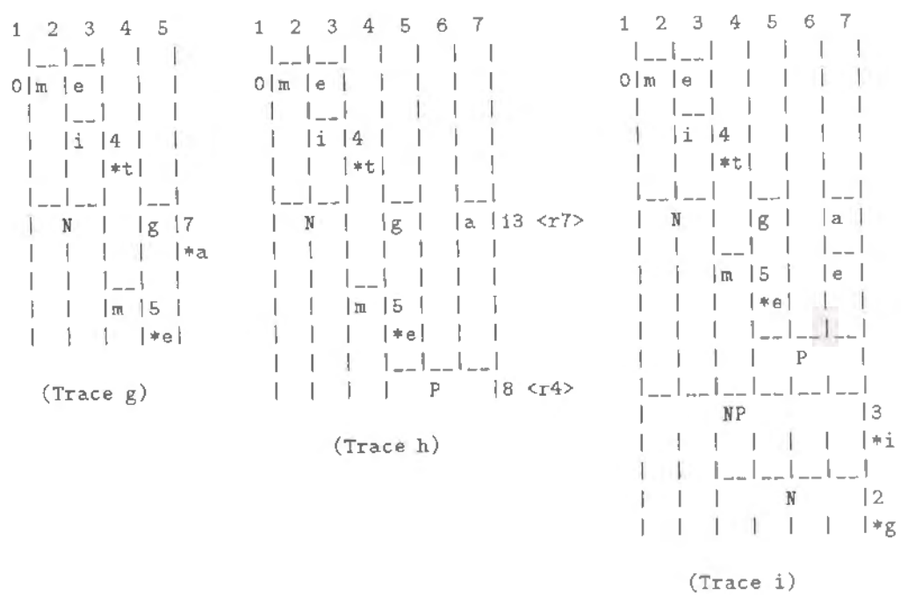


Figure 7.4: Example Trace g - i

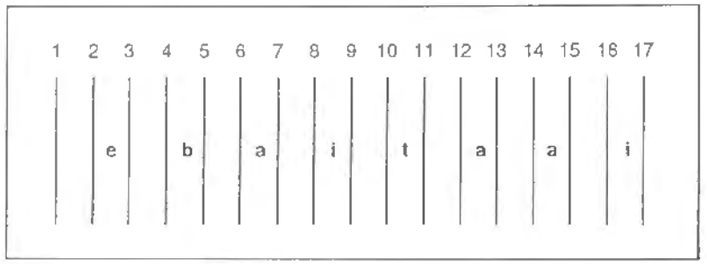


Figure 7.5: An input sequence of phonemes

Constraint 3: A shift action is not applied when the distance between the phoneme and the applied (non)terminal is more than 4. (This distance contains at least one real phoneme.)

Figure 7.4i shows the situation after "e" is applied.

The final configuration of the parser is represented in figure 7.6. Note here that the parser finds two successful parses: *megaitai* and *igaitai* ("I have a stomach ache").

7.3.3 Scoring and the Confnsion Matrix

There are two main reasons for scoring each candidate parse. The first is to prune the search space by discarding branches during the parse whose score is hopelessly low and therefore clearly incorrect. The second is to select the best sentence of multiple candidates by comparing their scores after parsing is complete.

Branches of the parse that consider fewer substituted/inserted/deleted phonemes should be given higher scores. This is a form of "least-deviant-first search" for ill-structured input, first introduced by [Fain *et al.*, 1985]. Whenever a branch of the parse handles a substituted/inserted/deleted phoneme, a specific penalty is applied to the branch. Unfortunately, the recognition device gives us neither the probability of each phoneme transition in the sequence nor the likelihood of finding substituted/inserted/deleted phonemes. Only the "best" phoneme sequence is given. Therefore we have to resort to a data structure called a *confusion matrix* for scoring purposes. A portion of the matrix is given in table 7.1.

The table shows part of the matrix produced by the manufacturer of the recognition device from sample word data. This matrix tells us, for example,

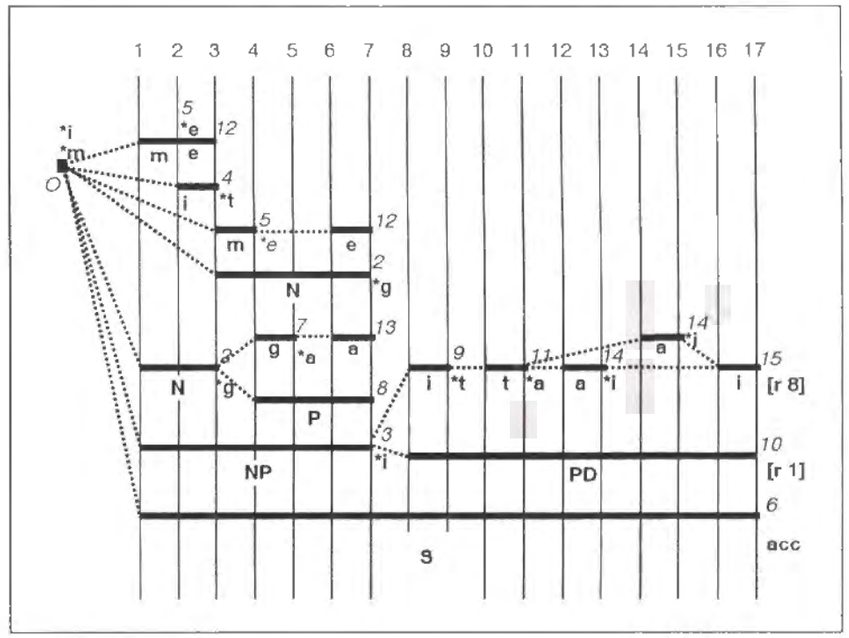


Figure 7.6: The final configuration of the parser

Output phoneme→ Input ↓ phoneme	/a/	/o/	/u/	/i/	/e/	/j/	/w/	...	(I)	(II)
/a/	93.8	1.1	1.3	0	2.7	0	0	...	0.9	5477
/o/	2.4	84.3	5.8	0	0.3	0	0.6	...	6.5	7529
/u/	0.3	1.8	79.7	2.4	4.6	0.1	0	...	9.7	5722
/i/	0.2	0	0.9	91.2	3.5	0.7	0	...	2.9	6158
/e/	1.9	0	4.5	3.3	89.1	0.1	0	...	1.1	3248
/j/	0	0	1.1	2.3	2.2	80.1	0.3	...	11.4	2660
/w/	0.2	5.1	5.8	0.5	0	2.6	56.1	...	11.2	428
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
(III)	327	176	564	512	290	864	212	...		

Table 7.1: A portion of a confusion matrix. (I) denotes the possibility of deleted phonemes; (II) the number of samples; and (III) the number of times this phoneme has been spuriously inserted in the given samples.

that if the phoneme /a/ is input, the device recognizes it correctly 93.8% of the time, misrecognizes it as /o/ 1.1% of the time, misrecognizes it as /u/ 1.3% of the time and so on. The column (I) says that the input is missed 0.9% of the time.

Conversely, if the phoneme /o/ is recognized by the device, there is a slight chance that the original input was /a/, /u/ or /w/, but no chance of it being /i/, /e/ or /j/, as can be seen from the table. The baseline *a priori* probability of the original input being /a/ is much higher than its being /w/. Thus, a substituted phoneme /w/ should be given a more severe penalty than /a/. A score for substituted phonemes can be obtained in this way, while deleted phonemes should be given a negative score, and inserted phonemes a zero or a negative score. With this technique, a score for a partial parse is calculated by summing the score of each constituent; the higher the score, the more likely the parse is correct.

Two methods have been adopted to prune partial parses by a score:

- Discarding the low-score, shift-waiting branches when a phoneme is applied; and
- Discarding the low-score branches in local ambiguity packing.

The former method, when strictly applied, is found to be very effective.

Note that the confusion matrix shows us only the phoneme-to-phoneme transition. It would seem that a broader-unit transition should also be considered, such as the tendency for the /w/ phoneme in 'owa' or 'owo' to be missed, the tendency for the very first /h/ sound of an input to be missed, and the frequent transformation to 'h@' of the 'su' sound in 'desuka'. In other words, a better confusion matrix can be constructed by considering a larger context—such as entire words.

7.3.4 Sample Runs

The actual output of the parser is shown in this section. The input phoneme sequence is "atomo ba itai" and the correct sequence is "atama ga itai"(which means "I have a headache."), which is produced as the top-score sentence of all parses. The frame-structure output after each parse is the meaning of the sentence. This meaning is extracted in the same way as the CMU's machine translation system does [Tomita and Carbonell, 1987b, Tomita *et al.*, 1988a].

7.4 Sphinx-LR

Sphinx is a state-of-the-art HMM speech recognizer developed at CMU [Lee, 1988]. It is being tested as a front end to a machine translation system. The grammar compiler produces three knowledge files used by Sphinx-LR from the context-free grammar. They are the pure context-free grammar rules, the LR parsing table and the HMM net file, as depicted in figure 7.8.

The grammar compiler expects an input grammar to be in the same format as the parser. Thus, when the user develops a grammar for Sphinx-LR, he or she can test and debug the grammar with typed input sentences using the standard LR parser. Once the grammar is debugged, it can be fed into the Sphinx-LR grammar compiler.

As already noted, the analysis grammar for the translation system is written in augmented context-free grammar, while basic Sphinx uses only a bigram grammar to reduce perplexity and thereby constrain the search process. Since the bigram grammar is generally much looser than the context-free grammar, Sphinx will output many incorrectly recognized ungrammatical sentences, which cannot be handled by the translation system. It is therefore desirable to constrain Sphinx's search process with the same grammar as the analysis grammar in the translation system so that the probability

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

1: (104) A<2-3> T<4-5> A<6-7#8> M<8-9> A<10-11#8> G<12-13#8> A<14-15>
         I<16-17> T<18-19> A<20-21> I<22-23>

((:MOOD DEC) (SEM *HAVE-A-PAIN441)
 (OBJ((:WH -) (CASE GA) (SEM *HEAD) (ROOT ATAMA))) (CAUSATIVE -) (OBJ-CASE GA)
 (SUBJ-CASE GA) (SUBCAT 2ARG-GA) (CAT ADJ) (:TIME PRESENT) (ROOT ITAI))

2: (94) A<2-3> S<4-5#8> A<6-7#8> M<8-9> A<10-11#8> D<12-13#8> E<14-15#8>
        I<16-17> K<18-19#8> O<20-21#8> U<22-23#8>

((:MOOD DEC) (SEM *PTRANS453)
 (PPADJUNCT
  ((PART MADE) (SEM *TIME) (ROOT *TIME) (:DAY-SEGMENT ((:CFNAME *MORNING)))))
 (SUBJ-CASE GA) (CAUSATIVE -) (PASSIVE -) (SUBCAT INTRANS)
 (:TIME (*OR* PRESENT FUTURE)) (SIYOU +) (CAT V) (ROOT IKU))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 7.7: Sample Outputs of the Parser

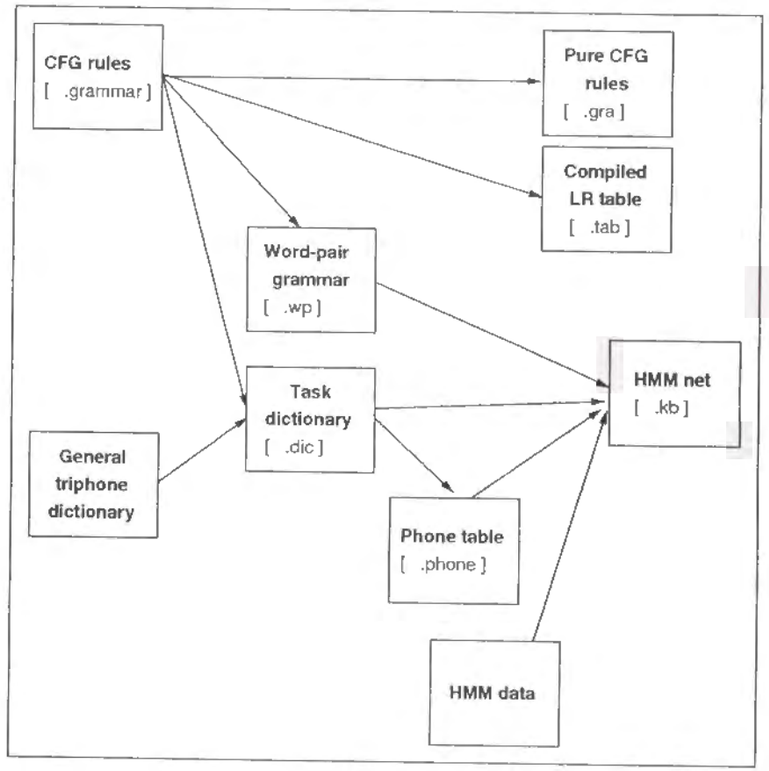


Figure 7.8: Sphinx-LR's compiled knowledge files

of correct recognition is increased and all Sphinx output can be handled by the translation system. A side benefit of more constrained search is faster speech recognition. Therefore, grammar-constrained speech recognition provides net benefits on all counts. The addition of semantic constraints improves performance beyond that of context-free grammars.

7.4.1 The HMM-LR Method

A technique called "HMM-LR" [Hanazawa *et al.*, 1990, Kita *et al.*, 1989b], has been developed to integrate Hidden Markov Models and GLR parsing. This subsection gives a description of the HMM-LR method. We assume here that the HMM recognizer applies at the individual phone level, although it can also be applied at the syllable or word levels.

In standard LR parsing, any parser action (*shift*, *reduce*, *accept* or *error*) is determined by using the current parser state and the next input symbol. This parsing mechanism is valid only for symbolic data (at any level of granularity), but cannot be applied simply to continuous data such as speech.

In HMM-LR, the LR parser is used as a language source model for word/phone prediction/generation. Thus we call the LR parser in next-symbol-set prediction mode the *predictive LR parser*. A phone-based predictive LR parser predicts next phones at each transition and generates possible sentences as phone sequences. The predictive LR parser determines next phones using the LR parsing table compiled from the specified grammar, and splits the parsing stack not only for grammatical ambiguity but also for alternative or confusible phone variation. Because the predictive LR parser uses context-free rules whose terminal symbols are phone names, the phonetic lexicon for the specified task is embedded in the grammar. A very simple example of a context-free grammar rules with a phonetic lexicon is given here:

```
(a)  S     -->   NP VP
(b)  NP    -->   DET N
(c)  VP    -->   V
(d)  VP    -->   V NP
(e)  DET   -->   /z/ /a/
(f)  DET   -->   /z/ /i/
(g)  N     -->   /m/ /ae/ /n/
```

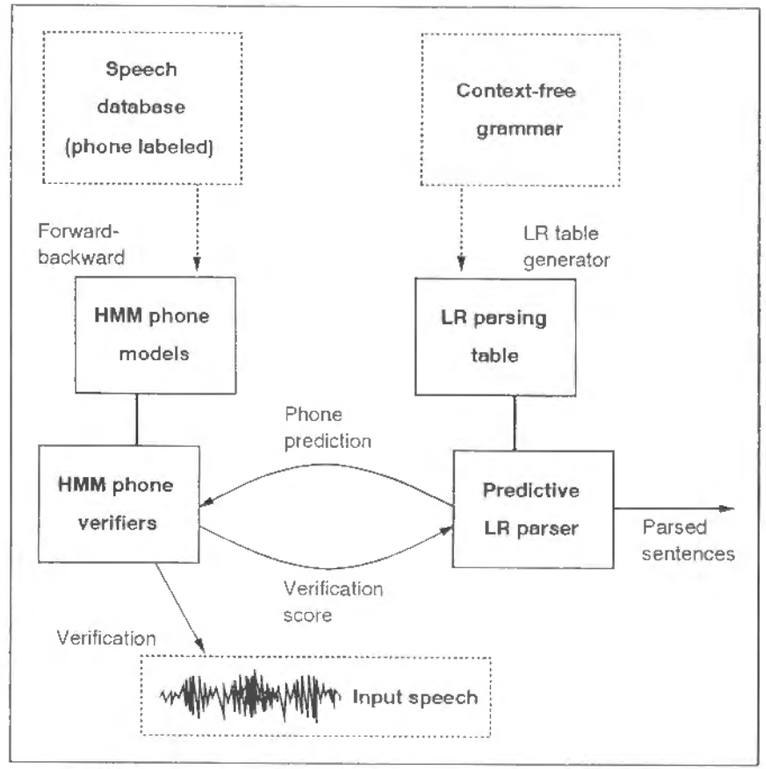



Figure 7.9: Schematic diagram of HMM-LR speech recognizer

(h) N --> /ae/ /p/ /a/ /l/

(i) V --> /iy/ /ts/

(j) V --> /s/ /ih/ /ng/ /s/

Here, rule (e) indicates the definite article *the* pronounced /z/ /a/ before consonants, while rule (f) indicates the *the* pronounced /z/ /i/ before vowels. Rules (g), (h), (i) and (j) phonetically indicate the words *man, apple, eats* and *sings*, respectively.

The HMM-LR continuous speech recognition system (see figure 7.9) consists of the predictive LR parser and HMM phone verifiers. First, the parser picks up all the phones predicted by the initial state of the LR parsing table and invokes the HMM models to verify the existence of these predicted phones. The parser then proceeds to the next state in the LR parsing table. During this process, all possible partial parses are constructed in parallel. The HMM phone verifier receives a *probability array* (see figure 7.10) which includes end-point candidates and their probabilities, and updates the array using a standard HMM probability calculation. This probability array is attached to each partial parse. Partial parses are pruned when their probability falls below a predefined threshold. In case more than one partial parse reaches completion, the one with highest probability is selected. For semantically constrained domains and clean speech signals, only one parse typically reaches completion.

7.4.2 The Integrated Speech-Parsing Algorithm

This section presents the algorithm for HMM-LR recognition more formally. The extension of the algorithm to produce parse trees during recognition is straightforward.

First, we introduce a data structure called a *cell*. A cell is a structure with information about one recognition candidate. The following items are kept in the cell:

- *LR parsing stack*, with information for parsing control.
- *Probability array*, which includes end-point candidates and their probabilities.

LR recognition proceeds as follows:

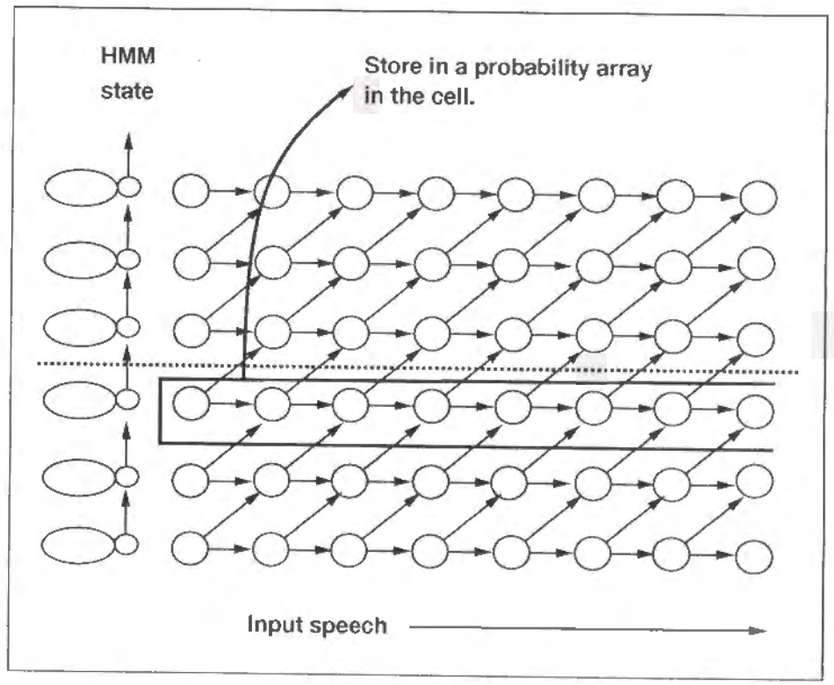


Figure 7.10: Stacking of a probability array

1. *Initialization*: Create a new cell C. Push the LR initial state 0 on top of the LR parsing stack of C. Initialize the probability array Q of C;

$$Q(t) = \begin{cases} 1 & t = 0 \\ 0 & 1 \leq t \leq T \end{cases}$$

2. *Ramification of cells*: Construct a set

$$\begin{aligned} S \quad = \quad & \{(C, s, a, x) | \exists C, s, a, x \ (C \text{ is a cell that is not accepted;} \\ & s \text{ is the state on top of the LR parsing stack of } C; \\ & \& \ x = ACTION[s, a] \ x \neq \text{“}error\text{”})\} \end{aligned}$$

For each element $(C, s, a, x) \in S$, perform the operations below. If set S is empty, parsing is completed.

3. If $x =$“*shift* s',” verify the existence of phone a. In this case, update the probability array Q of the cell C using the following computation:

$$\alpha_i(t) \quad = \quad \begin{cases} Q(t) & i = S_I \\ 0 & i \neq S_I \ \& \ t = 0 \\ \sum_j \alpha_j(t-1) a_{ji} b_{ji}(y_t) & i \neq 0 \ \& \ t > 0 \end{cases}$$

$$Q(t) \quad = \quad \begin{cases} 0 & t = 0 \\ \alpha_{S_F}(t) & t > 0 \end{cases}$$

If $\max_{1 \leq t \leq T} Q(t)$ is below a certain threshold, cell C is abandoned. Otherwise push s' on top of the LR parsing stack of cell C.

4. If $x =$ “*reduce* $A \rightarrow \beta$,” pop $|\beta|$ symbols off the LR parsing stack and push $GOTO[s', A]$ where s' is the current state on top of the stack.

5. If $x =$ “*accept*” and $Q(T)$ exceeds a certain threshold, cell C is accepted. If not, cell C is abandoned.

6. Return to 2.

Recognition results are kept in accepted cells. In general, many recognition candidates could exist, and it is possible to rank these candidates using $Q(T)$ of each cell.

In practice, however, the following two refinements of the above HMM-LR algorithm are useful:

1. Using beam-search technique:
The *beam-search* technique was first used in the HARPY speech recognition system [Lowerre, 1976, Lowerre and Reddy, 1980]. It is a modification of the breadth-first search technique, in which a group of near-miss alternatives around the best path are selected and processed in parallel, rather than retaining all candidates. The beam-search technique reduces search cost significantly and maintains accuracy. Generally, the set S constructed in step 2 in the algorithm is quite large. The beam-search technique is very useful for selecting the few most likely cells. The value $\max_{1 \leq t \leq T} Q(t)$ of each cell can be used as an evaluation score.

2. Using graph-structured stack:
The *graph-structured stack* is one of the key ideas in GLR parsing (described in chapter 2). In the above algorithm, when making a set S, virtual copies of the LR parsing stack are created. By using the graph-structured stack, it is not necessary to physically copy the whole stack. Copying only the necessary portion of the stack is sufficient and the amount of computation is reduced.

7.5 JANUS

A speech-to-speech translation system combining connectionist and symbolic processing strategies is being developed at CMU [Waibel *et al.*, 1991, Osterholtz *et al.*, 1992]. The system translates continuously spoken English speech input in the domain of conference registration dialogues into corresponding Japanese or German utterances. The system consists of three major components and integrates statistical, connectionist and knowledge-based approaches: the speech recognition (SR) component, the machine translation (MT) component and the speech synthesis (SS) component.

The MT-component employs several alternate processing strategies in parallel. To translate spoken language from one language to another, the analysis of spoken sentences, that suffer from ill-formed input and recognition errors is most certainly the hardest part. Based on the list of N-best hypotheses delivered by the recognition engine, we can now attempt to select and analyze the most plausible sentence hypothesis in view of producing and accurate and meaningful translation. Two goals are central in this attempt: *high fidelity* and *accurate translation* wherever possible, and *robustness* or *graceful degradation*, should attempts for high fidelity translation fail in face

of ill-formed or misrecognized input. At present, three parallel modules attempt to address these goals: 1) an LR-parser based syntactic approach, 2) a semantic pattern based approach and 3) a connectionist approach. The most useful analysis from these modules is mapped onto a common Interlingua, a language independent, but domain-specific representation of meaning. The analysis stage attempts to derive a high precision analysis first, using a strict syntax and domain specific semantics. Connectionist and/or semantic parsers are currently applied as back-up, if the higher precision analysis fails. The Interlingua ensures that alternate modules can be applied in a modular fashion and that different output languages can be added without redesign of the analysis stage.

7.5.1 Speech Recognition with Linked Predictive Neural Networks

Speech recognition is provided by a connectionist, continuous, large-vocabulary system using linked predictive neural networks (LPNN) [Tebelskis and Waibel, 1990]. In this system, neural networks are employed as predictors of speech frames, maintaining a pool of such networks as phoneme models. High-level algorithms are used to connect these networks into sequences corresponding to the phonetic spellings of words. With this linking of phonemic networks, the system is vocabulary independent and is applicable to large-vocabulary recognition.

Figure 7.11 illustrates the basic idea of signal prediction behind an LPNN network. To predict the next frame of the speech signal, K, contiguous speech frames are passed through a hidden layer of units in the network (shown as a triangle in the figure). The predicted frame is then compared to the actual frame, with the difference between the two indicating how good a model the network is for that segment of speech. If the network is taught to make relatively good predictions with respect to a particular phoneme (say /a/), then it is effectively an /a/ phoneme recognizer. In this way, a collection of phoneme recognizers can be obtained, with one model per phoneme. Each phoneme is actually modeled by a total of three subnetworks corresponding to the beginning, middle and end of the phoneme; the sequentiality of these constituent subnetworks being enforced by the LPNN architecture.

To allow for word recognition, each word is associated with a "linkage pattern" which is a logically constrained sequence of models corresponding to the phonetic spelling of the word. For instance, suppose the phonemes

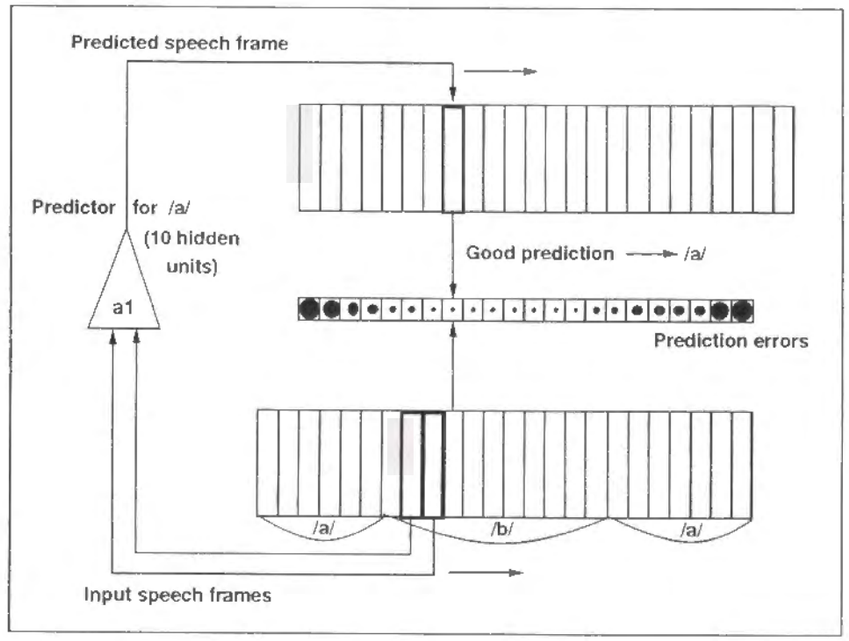


Figure 7.11: Modeling a phoneme by signal prediction

/a/ and /b/ are modeled by the sequence of networks a1,a2,a3, and b1,b2,b3 respectively, then the word *aba* is represented by the network linkage pattern: a1,a2,a3,b1,b2,b3,a1,a2,a3. Note that multiple occurrences of networks (like a1,a2 and a3) are linked together; this allows LPNN to model phonemes from varying contexts and to recognize words that were not in the training set.

We briefly describe a three-step training algorithm of the LPNN on a word:

1. *Forward pass*: For each input speech frame at time t, the frames at time $t-1$ and $t-2$ are fed into all the networks that are linked into this word. Each of these nets then makes a prediction of frame(t), and the prediction errors are computed and stored in a matrix.

2. *Alignment step*: Dynamic programming is applied to the prediction error matrix to find the optimal alignment between the speech signal and the phoneme models.

3. *Backward pass:* Errors are propagated backward along the alignment path. For each frame, the error is back-propagated into the network that best predicted the frame according to the alignment. Note that this alignment-controlled back-propagation causes each subnetwork to specialize on a different section of speech, resulting eventually in a model for each phoneme.

The training is repeated for all the words in the training vocabulary. A variation on the standard LPNN architecture has demonstrated a positive effect on the modeling performance: it allows a limited number of alternate models (say two or three) for each phoneme to account for the different characteristics of the phonemes in different contexts

In the experiment described in [Tebelskis and Waibel, 1990], recognition rates of 94% for a 234-word Japanese vocabulary of acoustically similar words, and 90% for a larger vocabulary of 924 words, are obtained.

7.5.2 Sentence Analysis with Generalized LR Parsing

The first step of the translation process is syntactic parsing with the GLR Parser/Compiler version 8-4 [Tomita and Carbonell, 1987a]. A grammar with about 455 rules for general colloquial English is written in a Pseudo Unification formalism [Tomita, 1990b], that is similar to Unification Grammar and LFG formalisms (see chapter 3 and appendix A for more about the


```
(HELLO IS THIS THE CONFERENCE OFFICE $)

;++++ GLR Parser running to produce English structure ++++

 (1) ambiguities fouud and took 1.164878 seconds of real time

(((PREV-SENTENCES ((COUNTER 1) (MOOD *OPENING)
                               (ROOT *HELLO)))
   (COUNTER 2)
   (MOOD *INTERROGATIVE)
   (SUBJECT ((AGR *3-SING) (ROOT *THIS)
                           (CASE (*OR* *NOM *OBL))))
   (FORM *FINITE)
   (PREDICATE
     ((DET ((ROOT *THE) (DEF *DEF))) (AGR *3-SING)
                                     (ANIM *-)
                                     (A-AN *A)
                                     (ROOT *CONFERENCE-OFFICE)))
   (AGR *3-SING)
   (SUBCAT *SUBJ-PRED)
   (ROOT *COPULA)
   (TENSE *PRESENT)))
```

Figure 7.12: Example F-Structure

GLR Parser/Compiler and its Pseudo Unification formalism). Figure7.12 shows the result of syntactic parsing of the sentence "Hello is this the conference office".

Modifications have been made to make the Generalized LR Parser more robust against ill-formed input sentences, using the *GLR* algorithm*, which is described in chapter 6. In case the standard parsing procedure fails to parse an input sentence, the parser nondeterministically skips some word(s) in the sentence, and returns the parse with fewest skipped words. In this mode, the parser will return some parse(s) with any input sentence, unless no part of the sentence can be recognized at all.

In the example in figure 7.13, the input sentence "Hello is this is this the office for the AI conference which will be held soon" is parsed as "Hello is this the office for the conference" by skipping 8 words. Because the analysis grammar or the interligua does not handle the relative clause "which will be

```
Input sentence :
(hello is this is this the AI conference office which will be held soon $))

Parse of input sentence :
  (HELLO IS THIS THE CONFERENCE OFFICE $)

Words skipped : ((IS 2) (THIS 3) (AI 7) (WHICH 10) (WILL 11)
                 (BE 12) (HELD 13) (SOON 14))
```

Figure 7.13: Example for robust parsing

held soon", 8 is the fewest possible words to skip to obtain a grammatical sentence which can be represented in the interligua. In the Generalized LR parsing, an extra procedure is applied every time a word is shifted onto the Graph Structured Stack. A heuristic similar to beam search makes the algorithm computationally tractable. See chapter 6 for more on the GLR* parsing algorithm.

When the standard GLR parser fails on all of the 20 best sentence candidates, this robust GLR* parser is applied to the best sentence candidate.

7.5.3 The Interlingua

This result, called "syntactic f-structure", is then fed into a mapper to produce an Interlingua representation. For the mapper, we use a software tool called TRANSKIT [Tomita *et al.*, 1988b] (see appendix B). A mapping grammar with about 300 rules is written for the Conference Registration domain of English.

Figure 7.14 is an example of Interlingua representation produced from the sentence "Hello is this the conference office". In the example, "Hello" is represented as speech-act *ACKNOWLEDGEMENT, and the rest as speech-act *IDENTFY-OTHER.

The JANUS interlingua is tailored to dialog translation. Each utterance is represented as one or more speech acts. A speech act can be thought of as what effect the speaker is intending a particular utterance to have on the listener. Our interlingua currently has eleven speech acts such as request direction, inform, and command. For purposes of this task, each sentence utterance corresponds to exactly one speech act. So the first task in the


```
((PREV-UTTERANCES ((SPEECH-ACT *ACKNOWLEDGEMENT) (VALUE *HELLO)))
  (TIME *PRESENT)
  (PARTY
    ((DEFINITE +) (NUMBER *SG)
                  (ANIM -)
                  (TYPE *CONFERENCE)
                  (CONCEPT *OFFICE)))
  (SPEECH-ACT *IDENTIFY-OTHER))
```

Figure 7.14: Example: Interlingua Output

mapping process is to match each sentence with its corresponding speech act. In the current system, this is done on a sentence by sentence basis. Rules in the mapping grammar look for cues in the syntactic f-structure such as mood, combinations of auxilliary verbs, and person of the subject and object where it applies. In the future we plan to use more information from coutext in determining which speech act to assign to each sentence.

Once the speech act is determined, the rule for a particular speech act is fired. Each speech act has a top level semantic slot where the semantic representation for a particular instance of the speech act is stored during translation. This semantic structure is represented as a hierarchical concept list which resembles the argument structure of the sentence. Each speech act rule contains information about where in the syntactic structure to look for constituents to fill thematic roles such as agent, recipient, and patient in the semantic structure. Specific lexical rules map nouns and verbs onto concepts. In addition to the top level semantic slot, there are slots where information about tone and mood are stored. Each speech act rule contains information about what to look for in the syntactic structure in order to know how to fill this slot. For instance the auxiliary verb which is used in a command determines how imperative the command is. For example, 'You must register for the conference within a week' is much more imperative than 'You should register for the conference within a week'. The second example leaves some room for negotiation where the first does not.

7.5.4 Sentence Generation

The generation of target language from an Iuterlingua representation involves two steps. Figure 7.15 shows sample traces of German and Japanese,

from the Interlingua in figure 7.14.

First, with the same TRANSKIT used in the analysis phase, Interlingua representation is mapped into syntactic f-structure of the target language.

There are about 300 rules in the generation mapping grammar for German, and 230 rules for Japanese. The f-structure is then fed into sentence generation software called GENKIT [Tomita *et al.*, 1988b] to produce a sentence in the target language. A grammar for GENKIT is written in the same formalism as the Generalized LR Parser: phrase structure rules augmented with pseudo unification equations. Detailed description of GENKIT and TRANSKIT can be found in appendix B.

The GENKIT grammar for general colloquial German has about 90 rules, and Japanese about 60 rules. Software called MORPHE is also used for morphlogical generation for German.

7.5.5 Semantic Pattern Based Parsing

A human-human translation task is even harder than human-machine communication, in that the dialog structure in human-human communication is more complicated and the range of topics is usually less restricted. These factors point to the requirement for robust strategies in speech translation systems.

Our robust semantic parser combines frame based semantics with semantic phrase grammars. We use a frame based parser similar to the DYPAR parser used by Carbonell, et al. to process ill-formed text[Carbonell and Hayes, 1984], and the MINDS system previously developed at CMU[Young *et al.*, 1989]. Semantic information is represented in a set of frames. Each frame contains a set of slots representing pieces of information. In order to fill the slots in the frames, we use semantic fragment grammars. Each slot type is represented by a separate Recursive Transition Network, which specifies all ways of saying the meaning represented by the slot. The grammar is a semantic grammar, non-terminals are semantic concepts instead of parts of speech. The grammar is also written so that information carrying fragments (semantic fragments) can stand alone (be recognized by a net) as well as being embedded in a sentence. Fragments which do not form a grammatical English sentence are still parsed by the system. Here there is not one large network representing all sentence level patterns, but many small nets representing information carrying chunks. Networks can "call" other networks, thereby significantly reducing the overall size of the system. These networks are used to perform pattern matches against input


```
;++ TransKit rules being applied to produce G structure ++

((PREV-SENTENCES ((VALUE HALLO) (ROOT LITERAL)))
         (ROOT SEIN) (CAT V) (PERSON 3)
         (SUBJECT
           ((CAT N) (CAS N) (DIST +) (LOC +) (PERSON 3)
            (NUMBER SG) (ROOT D-PRONOUN)))
         (NUMBER SG) (FORM FIN) (MOD IND) (TENSE PRES)
         (MOOD INTERROG)
         (PRED
           ((DET ((CAS N) (GENDER NEU)
                          (NUMBER SG)
                          (CAT DET)
                          (ROOT DER)))
             (CLASS SW) (NUMBER SG) (PERSON 3) (CAT N)
             (COMPOUND
               ((CAT N) (PL-CLASS PL3)
                        (SG-CLASS SG0)
                        (GENDER FEM)
                        (ROOT KONFERENZ)))
             (ROOT SEKRETARIAT) (PL-CLASS PL5) (SG-CLASS SG3)
             (GENDER NEU) (CAS N) (ANIM -))))

;++ GenKit rules being applied to produce German text ++

"HALLO , IST DORT DAS KONFERENZSEKRETARIAT  ?"

;++ TransKit rules being applied to produce J structure ++

((PREV-UTTERANCES
   ((FOR-REMOVE-DESU *IDENTIFY-OTHER) (VALUE MOSHIMOSHI)
                                      (ROOT *LITERAL)))
  (VTYPE MEISHI)
  (SUFF (*MULTIPLE* KA DESU))
  (PRED ((ROOT GAKKAIJIMUKYOKU) (CAT N)
                                (DEFINITE +)
                                (NUMBER *SG)
                                (ANIM -)))
  (ROOT COPULA))

;++ GenKit rules being applied to produce Japanese text ++

"MOSHIMOSHI GAKKAI JIMUKYOKU DESUKA"
```

Figure 7.15: Output language F-structure

word strings. This general approach has been described in [Ward, 1989, Ward, 1990].

The operation of the parser can be viewed as "phrase spotting". A beam of possible interpretations are pursued simultaneously. An interpretation is a frame with some of its slots filled. The RTNs perform pattern matches against the input string. When a phrase is recognized, it attempts to extend all current interpretations. That is, it is assigned to slots in active interpretations that it can fill. Phrases assigned to slots in the same interpretation are not allowed to overlap. In case of overlap, multiple interpretations are produced. When two interpretations for the same frame end with the same phrase, the lower scoring one is pruned. This amounts to dynamic programming on series of phrases. The score for an interpretation is the number of input words that it accounts for. At the end of the utterance, the best scoring interpretation is picked.

Our strategy is to apply grammatical constraints at the phrase level and to associate phrases in frames. Phrases represent word strings that can fill slots in frames. The slots represent information which, taken together, the frame is able to act on. We also use semantic rather than lexical grammars. Semantics provide more constraint than parts of speech and mnst ultimately be delt with in order to take actions. We believe that this approach offers a good compromise of constraint and robustness for the phenomena of spontaneous speech. Restarts and repeats are most often between phases, so individual phrases can still be recognized correctly. Poorly constructed grammar often consists of well-formed phrases, and is often semantically well-formed. It is only syntactically incorrect.

The parsing grammar was designed so that each frame has exactly one corresponding speech act. Each top level slot corresponds to some thematic role or other major semantic concept such as action. Subnets correspond to more specific semantic classes of constituents. In this way, the interpretation returned by the parser can be easily mapped onto the interlingua and missing information can be filled by meaningful default values with minimal effort.

Once an utterance is parsed in this way, it must then be mapped onto the interlingua discussed earlier in this section. The mapping grammar contains rules for each slot and subnet in the parsing gramar which correspond to either concepts or speech acts in the interlingua. These rules specify the relationship between a subnet and the subnets it calls which will be represented in the interlingua structure it will produce. Each rule potentially contains four parts. It need not contain all of them. The first part contains a default interlingua structure for the concept represented by a particular rule.

If all else fails, this default representation will be returned. The next part contains a skeletal interlingua representation for that rule. This is used in cases where a net calls multiple subnets which fill particular slots within the structure corresponding to the rule. A third part is used if the slot is filled by a terminal string of words. This part of the rule contains a context which can be placed around that string of words so that it can be attempted to be parsed and mapped by the LR system. It also contains information about where in the structure returned from the LR system to find the constituent corresponding to this rule. The final part contains rules for where in the skeletal structure to place interlingua structures returned from the subnets called by this net.

7.5.6 Connectionist Parsing

The connectionist parsing system PARSEC [Jain, 1991] is used as a fallback module if the symbolic high precision one fails to analyze the input. The important aspect of the PARSEC system is that it learns to parse sentences from a corpus of training examples. A connectionist approach to parse spontaneous speech offers the following advantages:

1. Because PARSEC learns and generalizes from the examples given in the training set no explicit grammar rules have to be specified by hand. In particular, this is of importance when the system has to cope with spontaneous utterances which frequently are "corrupted" with disfluencies, restarts, repairs or ungrammatical constructions. To specify symbolic grammars capturing these phenomena has been proven to be very difficult. On the other side there is a "build-in" robustness against these phenomena in a connectionist system.

2. The connectionist parsing process is able to combine symbolic information (e.g. syntactic features of words) with non-symbolic information (e.g. statistical likelihood of sentence types). Moreover, the system can easily integrate different knowledge sources. For example, instead of just training on the symbolic input string we trained PARSEC on both the symbolic input string and the pitch contour. After training was completed the system was able to use the additional information to determine the sentence mood in cases where syntactic clues were not sufficient. We think of extending the idea of integrating prosodic information into the parsing process in order to increase the performance

of the system when it is confronted with corrupted input. We hope that prosodic information will help to indicate restarts and repairs.

The current PARSEC system comprises six hierarchically ordered (back-propagation) connectionist modules. Each module is responsible for a specific task. For example, there are two modules which determine phrase and clause boundaries. Other modules are responsible for assigning to phrases or clauses labels which indicate their function and/or relationship to other constituents. The top module determines the mood of the sentence.

Recent Extensions:

We applied a slightly modified PARSEC system to the domain of air travel information (ATIS). We could show that the system was able to analyze utterance like "show me flights from boston to denver on us air" and that the system's output representation could be mapped to a Semantic Query Language (SQL). In order to do this we included semantic information (represented as binary features) in the lexicon. By doing the same for the CR-task we hope to increase the overall parsing performance.
We have also changed PARSEC to handle syntactic structures of arbitrary depth (both left and right branching) [Polzin, in preparation].

the main idea of the modified PARSEC system is to make it auto recursive, i.e. in a recursion step n it will take its output of the previous step n-1 as its input. This offers the following advantages:

1. **Increased Expressive Power**: The enhanced expressive power allows a much more natural mapping of linguistic intuitions to the specification of the training set.

2. **Ease of learning**: Learning difficulties can be reduced. Because PARSEC is now allowed to make more abstraction steps each individual step can be smaller and, hence, is easier to learn.

3. **Compatibility**: Because PARSEC is now capable of producing arbitrary tree structures as its output it can be more easily used as a submodule in NLP-systems (e.g. the JANUS system). For example, it is conceivable to produce as the parsing output f-structures which then can be mapped directly to the generation component [Buø, in preparation].

7.5.7 System Integration

The system accepts continuous speech speaker-independently in either input language, and produces synthetic speech output in near real-time. Our system can be linked to different language versions of the system or corresponding partner systems via ethernet or via telephone modem lines. This possibility has recently been tested between sites in the US, Japan and Germany to illustrate the possibility of international telephone speech translation.

The minimal equipment for this system is a Gradient Desklab 14 A/D-converter, an HP 9000/730 (64 Meg RAM) workstation for each input language, and a DECtalk speech synthesizer.

Included in the processing are A/D conversion, signal processing, continuous speech recognition, language analysis and parsing (both syntactic and semantic) into a language independent interlingua, text generation from that interlingua, and speech synthesis.

The amount of time needed for the processing of an utterance, depends on its length and acoustic quality, but also on the perplexity of the language model, on whether or not the first hypothesis is parsable and on the grammatical complexity and ambiguity of the sentence. While it can take the parser several seconds to process a long list of hypotheses for a complex utterance with many relative clauses (extremely rare in spoken language), the time consumed for parsing is usually negligible (0.1 second).

For our current system, we have eliminated considerable amounts of communication delays by introducing socket communication between pipelined parts of the system. Thus the search can start before the preprocessing program is done, and the parser starts working on the first hypothesis while the N-hest list is computed.

7.5.8 Summary

In this section, we have discussed recent extensions to the JANUS system a speaker independent multi-lingual speech-to-speech translation system under development at Carnegie Mellon and Karlsruhe University. The components include an speech recognition using an N-best sentence search, to derive alternate hypotheses for later processing during the translation. The MT component attempts to produce a high-accuracy translation using precise syntactic and semantic analysis. Should this analysis fail due to ill-formed input or misrecognitions, a connectionist parser, PARSEC, and a semantic

parser produce alternative minimalist analyses, to at least establish the basic meaning of an input utterance. Human-to-human dialogs appear to generate a larger and more varied breadth of expression than human-machine dialogs. Further research is in progress to quantify this observation and to increase robustness and coverage of the system in this environment.

Chapter 8

Concluding Remarks

In this document, we have discussed issues in sentence analysis for speech translation. Successful integration of speech recognition and mashine translation for practical applications requires, in the author's opinion, the following technologies:

- Efficient parsing algorithm
- Practical and robust parser implementation
- Development of solid grammars
- Handling ill-formed and erratic sentences
- Use of probabilistic and statistical information

We have addressed those issues with a number of different approaches, and each approach has been described in a different chapter. All the approaches presented are those of the projects at Carnegie Mellon University, and there are many other approaches to the issues of integration of speech and translation.

Finally, we should not forget that there are two more very important issues which are not addressed in this document. They are, of course:

- Accurate, efficient and robust speech recognition
- Accurate, efficient and robust machine translation

Clearly, speech translation is a giant problem which cannot be solved by a single person or project. Successful collaboration among researchers,

projects and institutions appears to be essential. The author and his colleagues would be most delighted if some of the sentence analysis techniques could make a contribution to scientific advances of the human dream: speech translation.

Appendix A

GLR Parser/Compiler Version 8-4: User's Manual

The Generalized LR Parser/Compiler Version 8-4 is is based on the Generalized LR Parsing Algorithm, augmented by pseudo and full unification packages [1]. The Generalized LR Parser/Compiler V8-4 is implemented in Common LISP and no window graphics is used; thus the system is transportable, in principle, to any machines that support Common LISP.

Those who are interested in obtaining the software described in this document should contact:

Radha Rao
Business Manager
Center for Machine Translation
Carnegie Mellon University
Pittsburgh, PA15213, USA
rdr@nl.cs.cmu.edu

Many members of CMU Center for Machine Translation have made contributions to the development of the system. The runtime parser was implemented by Hiroyuki Musha, Masaru Tomita and Kazuhiro Toyoshima. Hideto Kagamida and Masaru Tomita implemented the compiler. The pseudo unification package and the full unification package have been implemented by Masaru Tomita and Kevin Knight, respectively. Steve Morrisson, Hideto

[1] This appendix is based on a previously published technical report [Tomita *et al.*, 1988b]. I would like to acknowledge the coauthors of the report, Marion Kee, Teruko Mitamura, and Hiroyuki Musha, whose contributions are included in this appendix.

Tomabechi, Eric Nyberg and Hiroaki Saito also made contributions in maintaining the system. Sample English grammars have been developed by Donna Gates, Lori Levin and Masaru Tomita. A sample Japanese grammar has been developed by Teruko Mitamura. A sample French grammar is being developed by John Velonis and Linda Schmandt. Other members who made indirect contributions in many ways include Koichi Takeda, Marion Kee, Sergei Nirenburg, Ralf Brown, and especially Jaime Carbonell, the director of the Center.

Parts of this appendix were written by Hiroyuki Musha, Teruko Mitamura, Kevin Knight and Marion Kee.

A.1 Getting Started

A.1.1 Introduction

The Generalized LR Parser/Compiler V8-4 is based on the Generalized LR Parsing Algorithm described in chapter 2, augmented by pseudo/full unification packages. The grammar used by this parser is basically a set of context-free phrase structure rules, where each rule is paired with a list of *equations*, as described in detail in Chapter A.2.

Grammar compilation is the key to this efficient parsing system. A grammar written in the correct format has to be compiled before being used to parse sentences. The context-free phrase structure rules are compiled into an *Augmented LR Parsing Table*, and the equations are compiled into LISP functions. The runtime parser cannot parse a sentence without a compiled grammar.

The Generalized LR Parser/Compiler provides two different kinds of unification packages: pseudo uuification and full unification. Readers who are not familiar with a unification-based analysis of language are referred to:

> Shieber, Stuart M. *An Introduction to Unification-Based Approaches to Grammar*, 1986, Center for the Study of Language and Information, Stanford University, Stanford, CA.

Full unification is the canonical unification, and suitable for theoretical (linguistic) projects. Pseudo unification is suitable for practical projects, as it is faster and has practical operators such as arbitrary LISP function calls, sacrificing some of the theoretical elegance of canonical unification. The user has a choice between these two modes, as described further in Chapter A.8.

A grammar is interpreted on a *character-by-character basis*, rather than a *word-by-word basis*; that is, terminal symbols of a grammar are characters, not words. It is so designed with a view to the use of this system for *unsegmented languages* such as Japanese, in which there are no boundary spaces in between words. As a result of this character-based feature, the lexical dictionary and the morphological rules can be written in the same formalism as the syntactic rules.

A.1.2 A Sample Script

To begin with, let us compile the toy grammar shown in Figure A.1, and parse sentences using the compiled grammar. This extremely simple grammar can parse the sentences, "john", "john with john", "john with john with


```
(<np> <==> (<np> <pp>)
  (((x0 pp) = x2)
   ((x0 np) = x1)))

(<pp> <==> (<p> <np>)
  (((x0 np) = x2)
   ((x0 p) = x1)))

(<p> <--> (w i t h)
  (((x0 root) = with)))

(<np> <--> (j o h n)
   (((x0 root) = john)))
```

Figure A.1: A Toy Grammar, *toy.gra*

john", and so on. A detailed description of the grammar format can be found in section A.2.

```
* (load "init")
```

When you load the system, a message like the following will appear to greet you. It may take a couple of minutes.

```
"******************************************************"
"**       The Generalized LR Parser/Compiler       **"
"**                 RT version 8.1                   **"
"**         Center for Machine Translation           **"
"**           Carnegie Mellon University             **"
"**      (c) 1986, 1987 All rights reserved          **"
"******************************************************"

* (compgra "toy")
```

This is the way to compile a grammar. Use the function COMPGRA with the stem of a grammar file name as its argument. A grammar file has to have the extension ".gra". Something like the following messages will appear during the compilation of a grammar:

```
 - Reading toy.gra
 - toy.gra read

************************************************
***** Start compiling toy.gra
 - Reading toy.gra
 - toy.gra read
*** Grammar Pre-processor started
*** Grammar Pre-processor done
*** LFG Compiler started
*** LFG Compiler done
*** LR Table Compiler started
 - converting grammar
 - there were 4                    rules
 - there were 4 really different rules
 - there were 10 symbols
 - there were 7 terminal symbols
 - there were 3 non terminal symbols
 - making augmented grammar
 - making all items
 - 18 items made
 - collecting all items
LR {  0}
LR {  1}
LR {  2}
LR {  3}
LR {  4}
LR {  5}
LR {  6}
LR {  7}
LR {  8}
LR {  9}
LR { 10}
LR { 11}
LR { 12}
 - the number of states is 13
 - generating parsing table
LR'{  0}
 - reforming goto table
*** LR Table Compiler done
 - Writing File toy.tab
 - File toy.tab written
 - Writing File toy.fun
```



```
 - File toy.fun written
 - Writing File toy.funload
 - File toy.funload written
***** Setting up the runtime parser

Parser Ready
NIL

* (p "john with john")
```

After the compilation, the compiled grammar is loaded and the parser is set up automatically. The system is ready to parse a sentence.

```
>john with john

 1 (1) ambiguity found and took 0.141601 seconds of real time

;**** ambiguity 1 ***
((NP ((ROOT JOHN)))
 (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))



* (p  "johnwithjo hnwith john")
```

*The parser parses a sentence character-by-character, rather than word-by-word. In fact, it ignores blank spaces. Thus, it can accept an oddly-spaced sentence like the above. To stop ignoring spuces, set the variable *IGNORE-SPACE* to be nil (see section A.4.*

```
>johnwithjo hnwith john

 2 (2) ambiguities found and took 0.544923 seconds of real time

;**** ambiguity 1 ***
((NP ((NP ((ROOT JOHN)))
      (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))
 (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))

;**** ambiguity 2 ***
((NP ((ROOT JOHN)))
 (PP
```

```
  ((P ((ROOT WITH)))
   (NP ((NP ((ROOT JOHN)))
        (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))))))


* (p  "john with john with john with john")

>john with john with john with john

 3 (3) ambiguities found and took 0.843750 seconds of real time
```

*In case an input sentence is ambiguous, it will produce all possible structures. This sentence is 5 ways ambiguous, but local ambiguity is packed (see subsection A.3.4), and only 3 top-level structures are produced. The symbol *OR* represents local ambiguity.*

```
;**** ambiguity 1 ***
((NP
  (*OR*
   ((NP ((NP ((ROOT JOHN)))
         (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))
    (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))
   ((NP ((ROOT JOHN)))
    (PP
     ((P ((ROOT WITH)))
      (NP ((NP ((ROOT JOHN)))
           (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))
))))))))
 (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))

;**** ambiguity 2 ***
((NP ((NP ((ROOT JOHN)))
      (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))
 (PP
  ((P ((ROOT WITH)))
   (NP ((NP ((ROOT JOHN)))
        (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))))))

;**** ambiguity 3 ***
((NP ((ROOT JOHN)))
```



```
(PP
 ((P ((ROOT WITH)))
  (NP
   (*OR*
    ((NP ((NP ((ROOT JOHN)))
          (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))
     (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN))))))
    ((NP ((ROOT JOHN)))
     (PP
      ((P ((ROOT WITH)))
       (NP ((NP ((ROOT JOHN)))
            (PP ((P ((ROOT WITH))) (NP ((ROOT JOHN)))))))
      )))))))
```

A.1.3 Basic Functions

Compiling a Grammar

This is the complete form of the function `compgra` [2]:

`compgra` *string* `&key :result-to-file :parser-ready`

This function compiles a grammar file whose name is *string*`.gra`, and produces files: *string*`.tab`, *string*`.fun` and *string*`.funload`. The compiled grammar is automatically loaded, and the parser is ready to parse a sentence.

The symbols `:result-to-file` and `:parser-ready` are keywords whose values may be set to `t` in order to perform certain actions, and to `nil` in order not to perform them. If `:result-to-file` is set to `nil`, then it sets up the compiled grammar only on memory; it does not produce files, thereby saving time. Thus, once you get out of LISP, the compiled grammar disappears. The default is `t`. Set it to `nil` if you know that your grammar has bugs:

```
* (compgra "toy" :result-to-file nil)
```

If `:parser-ready` is set to `nil`, then it does not set up the parser. The default is `t`. Set it to `nil` if you do not want to test your grammar immediately.

```
* (compgra "toy" :parser-ready nil)
```

[2] The CommonLISP symbol `&key` is a lambda-list keyword.

Loading a Compiled Grammar

(`loadgra` *string*)

This function loads a compiled grammar, and sets up the parser. If you have just run `compgra`, you do not have to run `loadgra`.

Parsing a Sentence

(`p` *string*)

This function actually parses the sentence "*string*", and produces structures. If a sentence is ambiguous, it produces all possible structures but shows only the first three structures.

(`p*` *list-of-strings*)
This function parses all the sentences in the list. It is useful when you test a grammar with a set of test seutences.

Compiling Further a Compiled Grammar

(`make-gra-fast` *string*)
This function further compiles a compiled grammar to make it even faster. It produces one large binary file. Use `loadgra` to load the binary grammar. `loadgra` first looks for a binary file, and if one exists, it loads the binary grammar. It can take a few hours, if your grammar is large; so use this only when your grammar has become reasonably stable.

A.2 Writing a Grammar

This section describes how to write a grammar. A grammar file must have a name with the extension ".gra". The grammar formalism used in the system is similar to Lexical Functional Grammar (LFG) and PATR-II. This manual assumes that the reader knows Lexical Functional Grammar (LFG) as described in the following reference:

> Kaplan, R. & J. Bresnan (1982) in Bresnan (ed.) *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, MA.

We have structured this section more as a reference guide than as a tutorial for writing LFG-style grammars, since our formalism follows LFG closely enough that tutorial information would likely be redundant, given that the user already knows LFG.

A.2.1 General Format of Grammar Rules

A grammar rule for the Generalized LR Parser/Compiler V8-4 consists of a context-free phrase structure rule followed by a list of equations. The list of equations is enclosed in parentheses and the entire grammar rule is enclosed in parentheses.

```
(  context-free phrase structure rule
   ( list of equations ))
```

The following is a simple grammar for the sentence *A bird flies.*

```
(<S> <==> (<NP> <VP>)
   (((x1 case) = nominative)
    ((x1 agreement) = (x2 agreement))
    ((x0 subj) = x1)
    (x0 = x2)))

(<NP> <==> (<DET> <N>)
   (((x1 number) = (x2 number))
```

```
    ((x0 definiteness) = (x1 definiteness))
    (x0 = x2)))

(<VP> <==> (<V>)
  ((x0 = x1)))

(<N> <--> (b i r d)
  (((x0 root) = bird)
   ((x0 number) = sg)
   ((x0 agreement) = 3sg)))

(<V> <--> (f l i e s)
   (((x0 root) = fly)
    ((x0 agreement) = 3sg)
    ((x0 tense) = present)))

(<DET> <--> (a)
  (((x0 root)  = a)
   ((x0 definiteness) = -)
   ((x0 number) = sg)))
```

A.2.2 Phrase Structure Rules

The context-free phrase structure rule consists of a left-hand side, an arrow, and a right-hand side. The left-hand side must be a single non-terminal symbol. The arrow is one of the following: <==>,<-->, <--, or <==. The difference among them is not significant, unless the same grammar is also to be used in a sentence generator [3]. Discussions on sentence generation are beyond the scope of this documentation. Use whichever you like if you are concerned only with parsing.

The right hand side is a list of non-terminal symbols and alphanumeric characters enclosed in parentheses. Remember that a grammar is written in a *character basis*, and therefore terminal symbols of a grammar are characters, not words (see subsection A.1.1). There must be spaces placed between the

[3] In that case, the double headed arrows mean that the rule can be used for generation and parsing. An arrow pointing to the left means that the rule can be used only for parsing, and an arrow pointing to the right, ==> or -->, means that the rule can be used only for generation.

elements of the right hand side. Although spaces must be used in defining rules, spaces are ignored in parsing. Thus, the rule TIME --> (i n t h e m o r n i n g) will parse "in the morning", "inthemorning", "inth emor nin g", and so on. Non-terminal symbols must be enclosed in angle brackets.

A.2.3 Equations

Equations for the Generalized LR Parser/Compiler V8-4 are very similar to LFG equations except that they use the variables x0, x1, x2, x3, etc. in place of the up-arrow and down-arrow. X0 takes the place of the up-arrow. It refers to the functional structure corresponding to the left hand side non-terminal of the phrase structure rule. X1 takes the place of the down-arrow referring to the first element of the right hand side of the phrase structure rule. X2 takes the place of the down-arrow referring to the second element of the right hand side of the phrase structure rule, and so on.

Here is an example comparing an LFG rule with a Generalized LR Parser/Compiler rule:

LFG RULE:

```
S -->       NP            VP
      (^ subj) = v       ^ = v
```

GENERALIZED LR PARSER/COMPILER RULE:

```
(<S> <==> (<NP>  <VP>)
  (((x0 subj) = x1)
   (x0 = x2)))
```

The left hand side of an equation is a *path*. A path is:

- A variable (e.g. x0, x1, etc.).
- A variable followed by any number of character strings separated by spaces (e.g. (x1 subj), (x0 agreement), (x2 xcomp subject)). The character strings may not include certain special characters such as the quotation mark. This type of path must be enclosed in parentheses.

The right hand side of an equation is:

- A path.
- A character string (e.g. foot, headache, hurt, 12), excluding some special characters such as the quotation mark.
- A list consisting of the word *OR* followed by any number of character strings (e.g. (*OR* nominative accusative), (*OR* past pastparticiple)).

Each equation is enclosed in parentheses. The following is a list of example equations.

```
(x0 = x1)

((x0 subj) = x1)

((x1 case) = (*OR* nominative accusative))

((x1 agreement) = (x2 agreement))

((x0 root) = foot)

((x2 subj number) = sg)
```

A.2.4 The Starting Symbol

The left hand side of the first rule of the grammar is the start symbol. The start symbol is defined by the grammar writer, using the same kind of equations which are used for defining phrase structure rules. Defining the start symbol allows the grammar writer to decide if the system will only parse inputs which are full sentences, or accept both sentences and sentence fragments (phrases), or only accept phrases. For example, in order to parse full sentences as well as noun phrase fragments, the grammar would have to begin with the following two equations:

```
(<start> <==> (<S>)
  ((x0 = x1)))
```



```
(<start> <==> (<NP>)
  ((x0 = x1)))
```

The left hand side of a start equation contains the non-terminal `<start>`, and the right hand side contains a single non-terminal symbol which designates some constituent of the phrase-structure grammar. The parser will take any given input and attempt to parse it as the sort of constituent specified on the right hand side of the first start rule. If the input does not match the rules in the grammar which define that kind of constituent, the parser will attempt to match the input to the right hand side of the second start rule, and so on. Note that if fragmentary phrases such as `<NP>` or `<VP>` are not specified in the start rules, then the parse will fail when the input consists only of such a fragment, even if the fragment parses normally when included as part of a full sentence.

In order for this feature to work properly, the start equations *must* be the first equations listed in the grammar. The parser always assumes that it will accept *only* whatever structure is specified in the left hand side of the first grammar rule. By making the left hand side of the first rule be the non-terminal `<start>`, and then defining `<start>` one or more times, the grammar writer can cause the parser to accept a customized set of structure types.

A.2.5 Commenting the Grammar

Any line that begins with a semi-colon (;) is treated as a comment.

A.2.6 Disjunctive Equations

The Generalized LR Parser/Compiler V8-4 allows the user to specify a disjunction over a set of equations. A disjunction consists of the word *OR* followed by any number of lists of equations.

```
SCHEMA FOR DISJUNCTION EQUATIONS:

(*OR*
  ( list-of-equations )
  ( list-of-equations )
```

.....)

An example disjunction (from a grammar for English) is presented below. In the example, suppose that x2 stands for a <VP> and x1 stands for an <NP>, which we will assume is the subject of the <VP>. The goal is to make sure that if the <VP> is in the present tense, it agrees (in number and person) with the <NP> which is its subject. Otherwise, the <VP> must be in the past tense.

This disjunction contains two lists of equations. The first list contains two equations and the second list contains one equation. The disjunction says that either x2's *time* feature has the value *present* and x1's *agreement* feature has the same value as x2's *agreement* feature or x2's *time* feature has the value *past*.

```
(*OR*
  (((x2 time) = present)
   ((x1 agreement) = (x2 agreement)))
  (((x2 time) = past)))
```

Disjunctions can be used to give the effect of an if-then-else construction. For example, we could think of the disjunction above as saying that if x2's time is present then x1's agreement equals x2's agreement. Otherwise, x2's time feature is past.

The disjunction shown below contains four lists of equations. (The example is from an English grammar rule which implements verb sequence constraints. x2 and x3 both refer to <VP>'s.) The disjunction says that either:

- x2's passive feature has the value plus *and* x3's form feature has the value pastpart OR
- x2's passive feature has the value minus *and* x2's progressive feature has the value plus *and* x3's form feature has the value prespart OR
- x2's passive feature has the value minus *and* x2's progressive feature has the value minus *and* x2's perf feature has the valne plus *and* x3's form feature has the value prespart OR
- x2's passive feature has the value minus *and* x2's progressive feature has the value minus *and* x2's perf feature has the value minus *and* x2's modal feature has the value plus *and* x3's form feature has the value inf.


```
(*OR*
  (((x2 passive) = +)
   ((x3 form) = pastpart))
  (((x2 passive) = -)
   ((x2 progressive) = +)
   ((x3 form) = prespart))
  (((x2 passive) = -)
   ((x2 progressive) = -)
   ((x2 perf) = +)
   ((x3 form) = pastpart))
  (((x2 passive) = -)
   ((x2 progressive) = -)
   ((x2 perf) = -)
   ((x2 modal) = +)
   ((x3 form) = inf)))
```

Here is an example of a rule using one of the disjunctive equations shown above.

```
(<S> <==> (<NP> <VP>)
  ((*OR*
    (((x2 time) = present)
     ((x1 agreement) = (x2 agreement)))
    (((x1 time) = past)))
   ((x0 subj) = x1)
   (x0 = x2)))
```

A.2.7 Pseudo Equations

The Generalized LR Parser/Compiler V8-4 has two different modes for unification implementation: PSEUDO unification or FULL unification. FULL unification is the standard unification. PSEUDO unification is suggested as an alternative way to implement the unification; it does not do the unification, but does something very close to it. The implementation of PSEUDO unification is simpler and more efficient. The user can choose the unification mode by setting the variable *UNIFICATION-MODE* to be either

'FULL or 'PSEUDO. The defualt is 'PSEUDO. More discussions on pseudo unification can be found in section A.8.

The following operators are only for the PSEUDO mode, and not available if you choose the FULL unification mode.

Constraint Equations

Constraint equations use the symbol =c in place of the plain equal sign. The meaning of a constraint equation is the same as in LFG. A regular equation causes unification or assignment of a value to a function, while a constraint equation only checks to make sure that the function has the intended value. If the function does not already have the intended value, the parse will fail.

Examples:

```
((x1 case) =c nominative)

((x1 case) =c (*OR* nominative accusative))

((x3 form) =c pastparticiple)
```

Negative Constraint Equations

The word *NOT* can be used on the right haud side of an equation to check to see if the value specified in the equation does not exist.

Example:

```
((x2 subcat) = (*NOT* intransitive))
```

The above equation shows that the value of (x2 subcat) should be something other than intransitive. If the value is intransitive, the parse will fail.

UNDEFINED and *DEFINED*

The special words *UNDEFINED* and *DEFINED* can be used on the right hand side of an equation. *UNDEFINED* makes sure that the left

hand side of the equation has *no* value, and *DEFINED* makes sure that the left hand side of the equation has a value. For example, the equation

```
(x1 negation) = *UNDEFINED*
```

checks x1's negation feature to make sure that it has no value. If (x1 negation) has a value at the point when the equation is encountered, the parse will fail. These checks are useful for languages such as Japanese, for example where it is necessary to make sure that only one component of a sentence bears a particular feature.

Note that the equation given above does not assign a value to (x1 negation); it only checks for the presence of a value.

Assigning Multiple Values

Multiple values can be assigned to a feature or register by using the greater-than sign (>) in place of the equal sign. If the following rule applies recursively, the pp-adjunct function will have several different values at the same time:

```
(<S> <==> (<S> <PP>)
  ((x0 = x1)
   ((x0 pp-adjunct) > x2)))
```

LISP codes in the Grammar Rules

Arbitrary LISP codes can be written on the right-hand side of an equation, using the arrow <=. For example, the rule below deals with building integers from digits encountered in an input sentence. Paths in the LISP code, (x1 ...) and (x2 ...), are treated as special functions that return the value of the path.

```
(<integer> <--> (<integer> <digit>)
((x0 <= (+ (x1 value)
   (* 10 (x2 value))))))
```

The power of arbitrary LISP code is often very useful in a practical application, as in the following example cases.

- We may want to do some kind of semantic processing or inference, in parallel to the syntactic parsing. In that case, we need a method of triggering outside programs, namely the arbitrary LISP function call.
- When the phrase "three hundred twenty five" is parsed, we want to have a "value" slot filled by the integer 325; in that case, some arithmetic operations are necessary.
- The indefinite article "an" can be put only in front of phrases that begin with a vowel. While it is not impossible to ensure this agreement strictly within the feature/value framework, it might be much easier to have a LISP program to check it.

Wild Card Charactor

The off-line parser accepts a wild card character. If the wild card (currently % is the wild card character) appears in the grammar, it matches any single character and its value becomes the character itself. For example, the following rule assigns the value of `<char-seq>` to the character matched with

```
(<char-seq> <==> (%)
   (((X0 value) = (X1 value))))
```

By including the following rule with some LISP functions, the `<char-seq>` will accept any sequence of alphabetic characters:

```
(<char-seq> <--> ( <char-seq> % )
  (((X0 value) <=
            (read-from-string (concatenate 'string
   (symbol-name (x1 value))
   (symbol-name (x2 value)))))))
```

A.2.8 The Morphological Rules

A grammar for The Generalized LR Parser/Compiler is written in a *character basis*. Taking advantage of this feature, the morphological rules can be written in the same formalism as the syntactic rules. Affixation of a word can be handled by writing a context-free phrase structure rule. For example, Japanese complex verb forms can include causative morphemes, passive morphemes, various aspectual markers, and tense. These morphemes are not

included in a lexicon, but they are made available in the course of parsing. Morphological information in the form of an assignment equation is assigned to the functional structure. An example of some morphological and lexical rules for Japanese is provided below:

```
input string: tabe-sase-rare-ta
      eat-caus-pass-past

lexical rule:
(<v-1dan> <--> (t a b e)
(((x0 root) = taberu)))

morphological rules:
(<v-1dan> <--> (<v-1dan> s a s e)
(((x1 passive) = *UNDEFINED*)
 ((x1 tense) = *UNDEFINED*)
         (x0 = x1)
 ((x0 causative) = +)))

(<v-1dan> <--> (<v-1dan> r a r e)
(((x1 tense) = *UNDEFINED*)
 (x0 = x1)
 ((x0 passive) = +)))

(<v-1dan> <--> (<v-1dan> t a)
((x0 = x1)
 ((x0 tense) = past)))
```

The above morphological rules can parse the following verb forms:

```
tabeta eat-PAST
tabesaseta eat-CAUS-PAST
taberareta eat-PASS-PAST
tabesaserareta eat-CAUS-PASS-PAST
```

On the other hand, they cannot parse the following ungrammatical verb forms:

```
*taberaresaseta eat-PASS-CAUS-PAST
*tabetasaserare eat-PAST-CAUS-PASS
*tabetararesase eat-PAST-PASS-CAUS
```

```
*taberaretasase eat-PASS-PAST-CAUS
*tabesasetarare eat-CAUS-PAST-PASS
*taberaresase eat-PASS-CAUS
*tabetarare eat-PAST-PASS
*tabetasase eat-PAST-CAUS
```

As we can see in the morphological rules shown above, *UNDEFINED* and *DEFINED* equations are convenient devices to prevent a rule from applying in an undesirable order. Morphological rules such as the above reduce the total number of rules needed, because there is no need to write a separate entry for each verb form.

A.2.9 Dictionary: The Lexical Rules

A dictionary can be defined as a set of lexical rules each of which is a lexical entry. The lexical rules can include all affixations of a word without the use of any morphological rules. Alternatively, the lexical rules can contain only root forms, and morphological rules may be used to define all the affixation. For instance, the stem of a given verb is parsed by the lexical rule for that verb, and the equation portion of the lexical rule assigns a dictionary form of the verb to the root function. A lexical rule may also contain some additional infomation, such as subcategorization for verbs, and gender and number agreement, which will be needed in a later stage of parsing. The following is an example of a lexical rule for the Japanese verb *arau* ('wash'):

```
(<5-dan-w> <==> (a r a)
(((x0 root) = arau)
 ((x0 cat) = V)
 ((x0 subcat) = trans)))
```

The grammar writer may define macros for the lexical rules in the same way as CommonLISP macros are defined. We use macro definitions so that we no longer need to write each lexical rule separately; typing every lexical rule by hand is a time consuming task and may cause unnecessary bugs in the rules. The use of macros can reduce the size of the grammar, since the lexicon may be stored in a separate file. More details on the use of macros will be found in section A.5.

A.3 Debugging a Grammar

This section shows how to debug your grammar for the Generalized LR Parser/Compiler version 8-4. There are three main kinds of LISP functions that can help your debugging process:

- The function `dmode` shows you the rule application during the parsing process.
- The trace function provided by the LISP system shows you the input and output of the function associated with a particular rule.
- Other functions display useful information `after` the parsing.

In the following three subsections, each of these three kinds of functions will be described. In subsection A.3.4, *ambiguity packing* (which makes the parser efficient but makes debugging difficult) is explained.

A.3.1 Dmode

`Dmode` enables you to see which rules are being applied or being killed while the parser is running. By entering `(dmode 1)` before you run the parser, you will see applied rules as well as the input text. Entering `(dmode 2)`, you will see not only applied rules but rules killed because the functions associated with the rules did not return any value.

Look at the following example (this example is also used in the explanations of some of the other functions):

```
* (dmode 2)
2
* (p "remove it")

>remove it
R
E
M
O
V
E
 rule #1242 EV-IBMF-458      <V>(14) --> REMOVE
```

```
 rule #  45 E-IBMF-45       <VP>(15) --> <V>(14)
 killed - rule # 63 E-IBMF-63  <ASPECT> --> <V>(14)
I
T
 rule # 219 E-IBMF-219      <PRO>(28) --> IT
 rule # 209 E-IBMF-209      <NP>(29) --> <PRO>(28)
 rule #  46 E-IBMF-46       <VP>(30) --> <V>(14) <NP>(29)
 rule #  29 E-IBMF-29       <IMP>(32) --> <VP>(30)
 rule #   5 E-IBMF-5        <START>(51) --> <IMP>(32)
      ...
```

In the above example, the first rule applied was

```
<V>  --> remove
```

The output shows that the rule number is 1242, the name of the function was `EV-IBMF-458`, and the node `<V>` was assigned number 14. The function name `EV-IBMF-458` also tells you that the definition of the rule is in the file "ev-ibm", and it is the 458th rule in the file.

The next line shows that the `<V> (14)` became a `<VP> (15)`. The line after that shows that the `<V> (14)` could not become `<ASPECT>` because the attached function, namely `E-IBMF-63`, did not return any value. Note here that the parser analyzes the structures in parallel so that all the possibilities are tested.

A.3.2 Trace Function

The `trace` function provided by the LISP system is useful for looking at the input and output of a function called when a particular rule is applied.

If you enter `(trace e-ibmf-45)`, you will see the value passed to the function, which had been assigned to the node `<V>` numbered 14, and the value returned by the function, which was assigned to the node `<VP>` numbered 15.

Tracing functions is especially useful when a rule supposed to be successful is killed. You will often find the cause of the failure by looking at the input value passed to the function.

A.3.3 Other Functions

After parsing a sentence, the parser keeps various information about the structure of the sentence, and you can retrieve it using the following functions. Before you call these functions, run the parser under `(dmode 1)` or `(dmode 2)`. Otherwise, you cannot tell what node number was assigned to each node.

`disp-tree`

Function *disp-tree* displays the tree structure obtained, as shown below:

```
* (disp-tree)
 <START>(51) --> <IMP>(32)
     <IMP>(32) --> <VP>(30)
         <VP>(30) --> <V>(14) <NP>(29)
             <V>(14) --> R E M O V E
             <NP>(29) --> <PRO>(28)
                 <PRO>(28) --> I T
*
```

It also accepts an optional argument n, where n is a node number. If you want to see the subtree under the NP, whose node number is 29, you will input `(disp-tree 29)`.

This function can only display one ambiguity; if there are multiple ambiguities, one of them is chosen at random. If you want to see another ambiguity, you have to input the node number of the root node whose structure you want to see. To obtain node numbers, use `(dmode 2)` and/or `(disp-nodes)`.

`disp-node-value` *integer*

The function `disp-node-value` displays the category, son (child) nodes, and the value of the node. If you want to see the value of the NP whose node number is 29, type `(disp-node-value 29)`. You will see:

```
* (disp-node-value 29)
category = <NP>
sons = ((28))
value = ((:PRO +) (ROOT PRO) (REF DEF) (CASE ...))
```

`disp-nodes`

The function `disp-nodes` shows all the nodes with their sons (children). The following is an example:

```
* (disp-nodes)
  14  <V> --> R E M O V E
  15  <VP> --> <V>14
  16  <IMP> --> <VP>15
      ...
```

`disp-def`

The function `disp-def` displays the original definition of the rule you wrote in the grammar file (which has the ".gra" suffix.) If you want to see the definition of the rule number 1242, you enter `(disp-def "ev-ibm" 458)` (this information is obtained using `dmode 2`; see subsection A.3.1). Note that because grammar rules can be written in separate files, you have to look at the function name (e.g. `EV-IBMF-458`) to figure out which file the definition of this rule in, and where the rule is located in that file.

A.3.4 Ambiguity Packing

In the off-line parser, local ambiguities are packed into one node, which makes the parser quite efficient. This feature, however, makes the debugging of the grammar somewhat difficult. This subsection tells you how to deal with packed ambiguities.

Look at the following example:

```
* (p "move it period")
>move it
M
O
V
E
 rule #1156 EV-IBMF-372      <V>(12) --> MOVE
 rule #1152 EV-IBMF-368      <V>(12, was 13) --> MOVE
 rule #  45 E-IBMF-45        <VP>(14) --> <V>(12)
   ...
```

Both the rules numbered 1156 and 1152 make a `<V>` node and here the parser invoked the ambiguity packing procedure. By looking at the output you can see what happened.

The parser first makes the `<V>` node numbered 12 by applying the rule 1156. Then, it makes the second `<V>` node numbered 13 by applying the rule 1152. Knowing that these categories are the same, the parser merges node 13 to node 12 and the `<V>` node numbered 12 is handled as if it were a single node, except that node 12 has the values of both the first `<V>` node and the second `<V>` node. This procedure is called ambiguity packing.

You can see the value of node 12 by typing `(disp-node-value 12)`, and the value of node 13 before it was merged to 12 by typing `(disp-node-value 13)`. If you want to print the tree structure of a top level node, e.g. `<START>`

`(56)`, which was merged to another node, e.g. `<START> (53)`, you type `(disp-tree 53)`.

A.4 Changing Parameters

This section lists important global variables, which turn some features of the parser on or off. These variables are either set to `t` or `nil`, or they take some specified kind of argument, as explained below. The user may change the value of a parameter by using `SETQ` with the variable name and the desired argument.

1. `*recover-from-failure*`. If you set this variable to `t`, the parser will try to recover even if it fails during the parsing process. While you are debugging, it is better to set this variable to `nil`, in order to see where the parser failed. The default is `t`.

2. `*ignore-space*`. This variable toggles the word-based/character-based mode of the parser. If you do not want to ignore spaces in the input sentence, set this variable to `nil`, to use the word-based mode. For instance, if you are testing rules for English, you can set `*ignore-space*` to `nil`. In the case of Japanese, you have to set it to `t`. The default is `t`. (Note that this variable cannot affect the fact that you must leave spaces between characters when entering terminal symbols in the grammar rules. It only affects how the parser treats spaces encountered in the input.)

3. `*max-ambiguity-display*`. This variable determines how many ambiguities will be displayed at the end of the parse. It takes a numerical argument. Without limiting this, you sometimes see more than 10 pages of output. The default is 3.

4. `*wild-card-character*`. You can change the wild card character which matches any input character, by changing which character this variable is set to. The default is `%`.

5. `*unification-mode*`. This variable selects either full unification or pseudo unification for the unification mechanism. The possible values of this parameter are `full` or `pseudo`. The default is `pseudo`. (For more discussion on the difference between full and pseudo unification, see section A.8.)

A.5 Using Macro in a Grammar

When many of the rules in a grammar have the same or similar patterns, we often want to define a template or a *macro*, and to represent those rules by giving parameters to the macro. With the Generalized LR Parser/Compiler V8-4, you can define a macro anywhere in a grammar file and instantiate it anywhere in the same file. The way to define and call a macro is exactly the same as the Common LISP macro definition (see the Common LISP manual[4] Readers not familiar with LISP will need to acquire some knowledge of Common LISP in order to use macros effectively, and to follow the examples in this section.) Macro definitions are especially useful for lexical rules, as seen in the examples below.

A.5.1 A Simple Example

```
(defmacro %p (wordlist)
 (if (atom wordlist) (setq wordlist (list wordlist)))
 (append-dolist (word wordlist)
  `(((<p> <--> ,(explode-string word)
      (((x0 root) = ,(root-symbol word)))))))

(%p ("in" "at" "on" "until" "instead of"))
```

Given that the functions `explode-string` and `root-symbol` are appropriately defined, this macro definition/call has exactly the same effect as writing the 5 grammar rules below:

```
(<p> <--> (i n) (((x0 root) = in)))
(<p> <--> (a t) (((x0 root) = at)))
(<p> <--> (o n) (((x0 root) = on)))
(<p> <--> (u n t i l) (((x0 root) = until)))
(<p> <--> (i n s t e a d o f) (((x0 root) = instead-of)))
```

[4] Steele, Guy L., et al., *Common LISP: The Language*, Digital Press, Digital Equipment Corporation, Bedford, MA, 1984.

A.5.2 Lexical Rules for English Nouns

This subsection contains an extensive commented example of the use of macros, in conjunction with LISP functions, to write lexical rules for handling the plural forms of English nouns. The user may wish to try using these macros in the parser environment. It may be easier to follow the examples in the manual after seeing them produce results in the working system.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Lexical Rules for English Nouns with a Macro
;;;  10/5/87 Masaru Tomita created
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;
;;;  Use with the following grammar rules.
;;;
;;; (<n> <-- (<nroot>)
;;;       ((x0 = x1)
;;;        (*OR*
;;;  (((x1 pl-form) = irreg))
;;;  (((x1 pl-form) = (*OR* s es uncount))
;;;   ((x0 num root) = singular)
;;;   ((x0 agr) = 3sg)))))
;;;
;;; (<n> <-- (<nroot> y)
;;;       (((x1 pl-form) = ies)
;;;        (x0 = x1)
;;;        ((x0 num root) = singular)
;;;        ((x0 agr) = 3sg)))
;;;
;;; (<n> <-- (<nroot> s)
;;;       (((x1 pl-form) = s)
;;;        (x0 = x1)
;;;        ((x0 num root) = plural)
;;;        ((x0 agr) = other)))
;;;
;;; (<n> <-- (<nroot> e s)
;;;       (((x1 pl-form) = es)
;;;        (x0 = x1)
;;;        ((x0 num root) = plural)
;;;        ((x0 agr) = other)))
;;;
;;; (<n> <-- (<nroot> i e s)
;;;       (((x1 pl-form) = ies)
```



```
;;;        (x0 = x1)
;;;        ((x0 num root) = plural)
;;;        ((x0 agr) = other)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;
;;;  This is a function definition to be used in the macro:
;;;  REGULAR-NOUN-PL-FORM takes a countable noun, and returns
;;;  ies, es or s, assuming it's regular.
;;;
(defun regular-noun-pl-form (word)
  (let ((rword* (reverse (explode-string word))))
   (cond ((and (eq (first rword*) 'Y)
       (not (member (second rword*) '(a i u e o))))
    'ies)
   ((member (first rword*) '(S H A I U O))
    'es)
   (t 's))))

;;;
;;;  The macro will be named %n.  It takes a word or a
;;;  list of words, with optional keywords, and returns
;;;  one or more grammar rules with appropriate equations.
;;;  Here are examples of the rules it will return for
;;;  each kind of noun it encounters:
;;;
;;;  CASE 1. Regular countable noun (e.g. BOOK).
;;;     (%n "apple")  is expanded to:
;;;   (<NROOT> <--> (A P P L E)
;;;  (((x0 root) = apple)
;;;   ((x0 count) = YES)
;;;   ((x0 a-an) = an)
;;;   ((x0 pl-form) = s)))
;;;     (%n "boss")  is expanded to:
;;;   (<NROOT> <--> (B O S S)
;;;  (((x0 root) = boss)
;;;   ((x0 count) = YES)
;;;   ((x0 a-an) = a)
;;;   ((x0 pl-form) = es)))
;;;     (%n "copy")  is expanded to:
;;;   (<NROOT> <--> (C O P) ; note Y is missing
;;;  (((x0 root) = copy)
;;;   ((x0 count) = YES)
;;;   ((x0 a-an) = a)
```

```
;;;   ((x0 pl-form) = ies)))
;;;
;;;  CASE 2. Irregular countable noun (e.g. MAN).
;;;     (%n "man" :pl-form "men")  is expanded to:
;;;   (<NROOT> <--> (M A N)
;;;  (((x0 root) = man)
;;;   ((x0 count) = YES)
;;;   ((x0 a-an) = a)
;;;   ((x0 pl-form) = irreg))
;;;   ((x0 num root) = singular)
;;;   ((x0 agr) = 3sg))
;;;   (<NROOT> <--> (M E N)
;;;  (((x0 root) = man)
;;;   ((x0 count) = YES)
;;;   ((x0 a-an) = a)
;;;   ((x0 pl-form) = irreg))
;;;   ((x0 num root) = plural)
;;;   ((x0 agr) = other))
;;;
;;;  CASE 3. Uncountable noun (e.g. WATER)
;;;  (%n "water" :count NO) is expanded to:
;;;   (<NROOT> <--> (W A T E R)
;;;  (((x0 root) = water)
;;;   ((x0 count) = NO)
;;;   ((x0 pl-form) = uncount))
;;;   ((x0 agr) = 3sg))
;;;
;;;  A word can be a list of words.
;;;  e.g. (%n ("water" "coffee" "air") :count no)
;;;
;;;
;;;
;;;   Definition of the macro itself (with internal comments):
;;;
(defmacro %n (wordlist &key a-an  (count 'YES) pl-form )
 (if (atom wordlist) (setq wordlist (list wordlist)))
 (cond
;;;
;;;  CASE 1. Regular countable nouns.
;;;
  ((and (eq count 'YES)(null pl-form))
   (append-dolist (word wordlist)
    `((<nroot> <--> ,(if (eq (regular-noun-pl-form word) 'ies)
        (butlast (explode-string word))
```



```
        (explode-string word))
    (((x0 root) = ,(root-symbol word))
 ((x0 count) = YES)
   ((x0 a-an) = ,(or  a-an (if (member
(first (explode-string word))
        '(a i u e o))
    'an 'a)))
 ((x0 pl-form) = ,(regular-noun-pl-form word))
 ))
     )))
;;;
;;;  CASE 2. Irregular countable nouns.
;;;
  ((and (eq count 'YES) pl-form)
   (append-dolist (word wordlist)
    `((<nroot> <--> ,(explode-string word)
    (((x0 root) = ,(root-symbol word))
 ((x0 count) = YES)
   ((x0 a-an) = ,(or  a-an (if (member
(first (explode-string word))
        '(a i u e o))
    'an 'a)))
 ((x0 pl-form) = irreg)
 ((x0 num root) = singular)
       ((x0 agr) = 3sg)
 ))
       (<nroot> <--> ,(explode-string pl-form)
    (((x0 root) = ,(root-symbol word))
 ((x0 count) = YES)
   ((x0 a-an) = ,(or  a-an (if (member
(first (explode-string word))
        '(a i u e o))
    'an 'a)))
 ((x0 pl-form) = irreg)
 ((x0 num root) = plural)
 ((x0 agr) = other)))
     )))
;;;
;;;  CASE 3. Uncountable nouns.
;;;
 ((eq count 'NO)
  (append-dolist (word wordlist)
   `((<nroot> <--> ,(explode-string word)
    (((x0 root) = ,(root-symbol word))
```

```
 ((x0 count) = NO)
 ((x0 pl-form) = 'uncount)
 ((x0 agr) = 3sg)
 )))))))
;;;
;;;  %n is now called with nouns from the doctor/patient domain:
;;;

(%n
 ("glass" "medicine" "tablet" "rubberband" "meal" "mouth" "bath"
  "knee" "finger" "had" "thumb" "pain" "sore" "fever" "head" "headache"
  "stomach" "drug" "allergy" "reaction" "ache" "back" "eye" "hangover"
  "bigtoe" "throat" "abdomen" "arm" "neck" "elbow" "chest"
  "aspirintablet" "leg" "hand" "stiffness"
 ))

(%n
 ("aspirin" "water" "milk" "saltwater" "liniment" "temperature" "nausea"
  "numbness"
 )
 :count NO)

(%n "foot" :pl-form "feet")
```

A.5.3 Lexical Rules for English Verbs

This subsection contains an extensive commented example of the use of macros, in conjunction with LISP functions, to write lexical rules for creating the correct analysis of English verb forms:

```
;;;;;
;;;;;    Macro Definition for English Verbs
;;;;;     5/26/87  Masaru Tomita
;;;;;

;;;  This macro requires the MAP-DOLIST macro, usually already
;;; defined in pseudo-unify.LISP

;;; Specification:
;;;  (%ev-list
;;;     (word {:PAST pastform}{:PASTPART pastpartform}
;;;       {:SYL-DOUBLE t-or-nil}{:OTHERS other-inf})
;;;     (word {:PAST pastform}{:PASTPART pastpartform}
```



```
;;;       {:SYL-DOUBLE t-or-nil}{:OTHERS other-inf})
;;; ....
;;;  )
;;; Note: "word", "pastform" and "pastpartform" can be either symbol or
;;;      string (for compound verbs). "other-inf" is either (slot value)
;;;      pair, or a list of (slot value) pairs.

;;; Example:
;;;  (%ev-list
;;; (accord)
;;;     (take :past took :pastpart taken)
;;;     ....
;;;  )
;;;     =====>
;;;  (<vroot> <-- (a c c o r d)
;;;     ((x0 <= '((root accord)
;;;    (valency trans)
;;;    (a-an an)
;;;    (morph-type other)))))
;;;  (<vroot> <-- (t a k)
;;;     ((x0 <= '((root take)
;;;    (valency trans)
;;;    (a-an a)
;;;    (morph-type e)))))
;;;  (<vroot> <-- (t o o k)
;;;     ((x0 <= '((root take)
;;;    (valency trans)
;;;    (a-an a)
;;;    (irreg-past +)
;;;    (form finite)
;;;    (tense ((root past)))))))
;;;  (<vroot> <-- (t a k e n)
;;;     ((x0 <= '((root take)
;;;    (valency trans)
;;;    (a-an a)
;;;    (irreg-pastpart +)
;;;    (form pastpart)))))
;;;

;;;
;;;  There are four values for MORPH-TYPE:
;;;    Y:  Ends with y, and the last char but one is not a vowel.
;;;     e.g. "copy", "supply", ... but not "play", "employ", ...
;;;    E:  Ends with e.  e.g. "hope", "type", ...
```

```
;;;    SH: Ends with a, i, o, u, s or h.  e.g. "pass", "finish", ...
;;;    OTHER: others.  e.g. "print", "employ", "write", ...
;;;      MORPH-TYPE is independent from whether the verb is regular or
;;;      irregular.
;;;
;;;  <VROOT> is:
;;;    root-word without the last "y" or "e", if MORPH-TYPE is Y or E.
;;;    root-word, if MORPH-TYPE is SH or OTHER.
;;;  E.g., <VROOT>'s of "copy", "hope" and "print" are (C O P), (H O P)
;;;  and (P R I N T), respectively.
;;;
;;;  IRREG-PAST and IRREG-PASTPART are defined "+", if a verb has an
;;;  irregular form for past and past participle from, respectively.
;;;  E.g., "write" has + in both IRREG-PAST and IRREG-PASTPART.
;;;
;;;  FORM is always undefined except for irregular forms, in which case
;;;  special rules are defined and FORM is defined.
;;;
;;;  SYL-DOUBLE is defined in case the last character must be doubled
;;;  in "ing" and "ed" forms. Otherwise it remains undefined.
;;;  E.g., "hit" has SYL-DOUBLE "t", "pop" has SYL-DOUBLE "p", and
;;;  "print" has SYL-DOUBLE undefined.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  The macro %ev does most of the work on the verb forms, but it
;;;  is not the top-level macro.  The top-level macro, which accepts
;;;  words in a user-friendly format and calls %ev on them, is
;;;  named %ev-list.

(defmacro %ev (word &key syl-double past pastpart others
 a-an (valency 'trans))

;;;   The following provisions are to tolerate errors.
;;;
  (if (symbolp word)       ;; If word is a symbol,
      (setq word (symbol-name word)))        ;; then make it a string.
  (if (and past (symbolp past))       ;; If past is a symbol,
      (setq past (symbol-name past)))        ;; then make it a string.
  (if (and pastpart (symbolp pastpart))       ;; If pastpart is a symbol
      (setq pastpart (symbol-name pastpart))) ;; then make it a string.
  (if (and others (atom (car others)));; If :OTHERS is not a list of
      (setq others (list others)))    ;; (slot value)'s, make it a list.
```



```
  (let* ((word* (explode-string word))
 (rword* (reverse word*))
   (root* (root-symbol word))
 (a-an (or a-an (if (member (first word*) '(A I U E O))
    'an 'a)))
 (morph-type (cond
((and (eq (first rword*) 'Y)
      (not (member (second rword*)
    '(A I U E O))))
 'Y)
((eq (first rword*) 'E)
 'E)
((member (first rword*) '(A I U O S H))
 'SH)
(t 'OTHER)))
 (irregular-rule-past
   (if past
       '((<vroot> <-- ,(explode-string past)
 ((x0 <= '((root ,root*)
   (form finite)
   (a-an ,a-an)
   (valency ,valency)
   ,@others))
  ((x0 tense root) = past))))))
(irregular-rule-pastpart
  (if pastpart
      '((<vroot> <-- ,(explode-string pastpart)
 ((x0 <= '((root ,root*)
   (form pastpart)
   (a-an ,a-an)
   (valency ,valency)
   ,@others)))))))
(regular-rule
      '((<vroot> <-- ,(case morph-type
  ((Y E)(butlast word*))
  ((SH OTHER) word*))
     ((x0 <= '((root ,root*)
(valency ,valency)
(a-an ,a-an)
,@(if morph-type '((morph-type ,morph-type)))
,@(if past '((irreg-past +)))
,@(if pastpart '((irreg-pastpart +)))
,@(if syl-double
 '((syl-double ,(first rword*))))))
```

```
,@others)))))

;;; let* body
    '(,@regular-rule ,@irregular-rule-past ,@irregular-rule-pastpart)))

;;;
;;;  Top-Level Macro
;;;
(defmacro %ev-list (&rest v-list)
  (append-dolist (v v-list)
(macroexpand (cons '%ev v))))

(%ev-list
   ("ache")
   ("aggravate")
   ("apply")
   ("become" :past "became" :pastpart "become")
   ("begin" :past "began" :pastpart "begun")
   ("bathe")
   ("breathe")
   ("burn")
   ("develop")
   ("drink" :past "drank" :pastpart "drunk")
   ("eat" :past "ate" :pastpart "eaten")
   ("feel" :past "felt" :pastpart "felt")
   ("hit" :past "hit" :pastpart "hit" :syl-double T)
   ("hurt" :past "hurt" :pastpart "hurt")
   ("lift")
   ("make" :past "made" :pastpart "made")
   ("move")
   ("put" :past "put" :pastpart "put" :syl-double T)
   ("rinse")
   ("show" :pastpart "shown")
   ("step" :syl-double T)
   ("swallow")
   ("take" :past "took" :pastpart "taken")
   ("talk")
   ("tell" :past "told" :pastpart "told")
   ("throb" :syl-double T)
   ("turn")
   ("upset" :past "upset" :pastpart "upset" :syl-double T)
   ("walk")
```



```
)
```

A.6 Compiling Lexical Files Separately

Sometimes, one finds it frustrating that a whole grammar has to be re-compiled every time a small change is made. It would be very nice if one could write a grammar in several separate sub-files, and compile only those sub-files in which changes have been made. This general idea, incremental compilation, is not fully implemented in the Generalized LR Parser/Compiler version 8-4; it can be done only with several constraints as described below.

- A grammar should consist of one main file and several sub-files.
- In the main file, all sub-files must be declared as in the following example.

  ```
  (@lex "sub1" "sub2" sub3")
  ```

  All sub-files, as well as the main file, must have a name with the extension ".gra".
- In each sub-file, the right hand side of every rule must consist entirely of terminal symbols. In other words, all rules in a sub-file have to be lexical rules.
- In each sub-file, the left hand sides of all rules must be identical. There must not exist rules such that

  ```
  (<N> <--> (....))
  ```

  and

  ```
  (<ADJ> <--> (....))
  ```

  are included in a single sub-file.

In sub-files, macros can be defined as described in the previous section. To compile a grammar, you simply use the `compgra` function over only its main file. `compgra` will automatically find and integrate sub-files. When changes are made in one or more files, `compgra` will re-compile only files in which changes have been made. It automatically checks to see which files have been changed since the last compilation of the grammar. Thus, all you have to do to include sub-files is to use `compgra` with the main file name.

A.7 Using Your Own Morph/Dictionary System

A.7.1 Introduction

We have described our parsing system as a character basis system (rather than word basis). That is, all terminal symbols in a grammar are characters (or letters), so that one can write morphological rules and dictionary in the same formalism as syntax. It is, however, entirely possible to use the Generalized Parser/Compiler for conventional word basis parsing, and to access your own separate system for morphology and dictionary. The following two subsections describe how to write a word based grammar, and how to incorporate your own morph/dictionary system.

A.7.2 Word-Based Parsing

You can write a grammar with word symbols as terminals, without changing anything to the system. For example, instead of writing
`(<N> <--> (b o o k) ....)`
you can simply write `(<N> <--> (BOOK) ....)`

where "BOOK" is one symbol. Such grammars can be compiled in exactly the same way as before. However, you have to use `parse-list`, rather than `p` for runtime parsing.

(`parse-list` *list-of-symbols*)
This function parses the sentence represented by the list of symbols followed by the character '$.

For example,

```
(parse-list '(A BIRD FLIES $))
```

A.7.3 User's Dictionary Look Up

You can call your own dictionary look up program and/or morphological analyzer using `'<=`, as in the following example.

```
(<N> <-- (%)
  ((x0 <= (diction (x1 value) 'noun)))
(<V> <-- (%)
  ((x0 <= (diction (x1 value) 'verb))))
 ...
```

In this example, the function `diction` is the user-defined function that takes a word symbol and its category, and returns an appropriate f-structure for the word. Recall that "%" is the wild card character that can match with any single symbol, and the symbol is put into the value slot of x1.


```
(S <==> (NP VP)
   (((x1 case) = nom)
    (*OR*
      (((x2 tense) = present)
       ((x1 agr) = (x2 agr)))
      (((x2 tense) = past)))
    (x0 = x2)
    ((x0 passive) = no)
    ((x0 subj) = (x1))))
```

Figure A.2: An Example Rule

A.8 Pseudo Unification and Full Unification

A.8.1 Introduction

The Generalized LR Parser/Compiler V8-4 supports two kinds of unification implementation: FULL unification and PSEUDO unification. The user can choose between them by setting the variable `*UNIFICATION-MODE*` to be either `FULL` or `PSEUDO`. Full unification is the canonical unification, and it behaves the same as other unifcation based systems such as PATR-II.

PSEUDO unification is an alternative approach to the unification implementation proposed by Tomita and Knight [Tomita and Knight, 1988]. It does not exactly do unification, but does something close to it. In fact, it produces the same results as full (canonical) unification most of the time, and it does not seem to present any problems in practical applications, such as natural language interfaces and machine translation. On the other hand, feature structures in pseudo unification are always simple trees rather than dags (directed acyclic graphs). Thus, the implementation of pseudo unification is much simpler, bypassing all the tough problems caused by dags.

A.8.2 Full Unification

Consider the following sample grammar rule of English (figure A.2). `x0`, `x1`, and `x2` are feature structures (graphs) with categories S, NP, and VP, respectively. The rule states constraints on the feature structures, e.g. the case of the NP must be nominative. In parsing, rules such as this one are used to construct larger constituents out of smaller ones. The rule itself can

be viewed as a feature structure with top-level features x0 x1, and x2. An equation such as "(x1 agr) = (x2 agr)" indicates that two features in the rule graph must share the same value. Once we have feature structures for an NP and VP, we combine them and unify them with the rule structure. Unification is a process which forms the union of two feature sets, and which detects value conflicts between them. Thus, the equations in the rule above serve two purposes: (1) to test features of the constituents, and (2) to build new structure. When the rule is applied, unification of the rule structure and the constituent structure takes place. For example, if the NP and VP have agr features with different values, the unification will fail; otherwise, unification will succeed, and the resulting feature structure will contain two features with a single common value. Note that they would not simply contain two values that are alike: they share a common value (in the sense of Lisp's EQ, not EQUAL). This sharing property, often called re-entrancy, is what makes graph representation necessary for feature structures. The result of applying the rule is a new feature structure, of category S, which may now be used as a constituent in another rule. In this case, whether or not the structure contains re-entrancy can affect later unifications.

A.8.3 Pseudo Unification

In pseudo unification, there is no re-entrancy. There may be two features with identical values (one value for one feature), but there never be two features that share a value. Therefore, a feature structure can be always represented as a *tree*, rather than a graph. This will make a drastic difference in terms of simplicity and efficiency, as discussed in the next subsectiou.

Let us assume that the algorithm we use for context-free parsing is bottom-up (possibly with top-down predictions); it combines constitutents in the right hand side of a rule into a new constituent of the left hand side. Each constituent has a feature structure which is a tree (or an atom). Equations in a rule are interpreted to construct a new feature structure for the new constituent from feature structures of right hand side constituents. Instead of viewing a rule itself as a feature structure, we interpret equations in the rule one by one from the top to the bottom. Each equation of the form, (xn ...) = (xm ...), is interpreted procedurally as follows.

1. Get the value of (xn ...).

2. Get the value of (xm ...).

3. Unify these two values.

4. If successful, store the result in both (xn ...) and (xm ...); If unsuccessful, die.

For example, consider the equation, (x1 agr) = (x2 agr), in figure A.2. Get the agreement value of NP (which may be a tree structure or an atom), and get the agreement value of VP. Unify them. If the unification fails, forget about applying this rule. If the unification returns a new value (which may be different from the original agreement values), put the new value into (x1 agr) and (x2 agr) as their new agreement values. Note that if the original agreement values are identical, then we do not have to put the new value. Also, observe that if one of the features is not defined, then it will act as a simple assignment of the value.

If an equation is of the form, (xn ...) = atom-value, then it is interpreted as follows.

1. Get the value of (xn ...).

2. Unify the value and the atom-value.

3. If successful, put the result to (xn ...); If unsuccessful, die.

Consider, for example, the equation, (x1 case) = nom, in figure A.2. Get the case value of NP, and unify it with nom. If the nnification fails, forget about applying this rule. If the unification returns a new value (which may be different from the original values if either or both of them are disjunctive), put the new value to (x1 case).

A.8.4 When Pseudo Unification Works Differently

Pseudo-unification can give results different from full-unification. The simplest case takes place within a single rule, such as in figure A.3.

Assume that the VP has no voice feature before the unification. Full-unification would produce a feature structure whose x0 feature contained a voice feature with value active. Pseudo-unification, on the other hand, would interpret the "(x0 = x2)" equation in the following way: tree-unify (x0) and x2, then store two separate copies of the result in x0 and x2. In that case, the third equation could not affect the x0 feature, and the x0 feature of the result graph would contain any voice feature at all.

A more complicated case where pseudo-unification behaves differently occurs between rule applications. Pseudo-unification, unlike full-unification,

```
(S <==> (NP VP)
   (((x0 subj) = x1)
    (x0 = x2)
    ((x2 voice) = active)))
```

Figure A.3: Counter Example I

cannot carry re-entrant structures across rule applications. This behavior can show up when modeling complex agreement phenonmena. Consider the rules in figure A.4.

Given the sentence "dogs run", we can apply the first two rules bottom up to get feature structures of categories N and V. The third rule can then apply, fixing the agreement features of the N and V to share the same value. (If the sentence had been "dogs runs", the unification would fail). The fourth rule then applies, but fails under full-unification – there can be only one gender feature, because there is only one agreement feature. Under pseudo-unification, however, the third rule behaves differently. The agreement features of the N and V are unified, and two copies are stored, one in the agr feature of the main structure, and one in the agr feature of the subject. When the fourth rule applies, it has no problem assigning different genders to the two different agreement features.

While there exist rules like above that pseudo unification does not handle properly, it does not necessarily mean that pseudo unification cannot handle certain linguistic phenomena that full unification can handle. In fact, there seem to always exist some way of rewriting rules so that pseudo unification can behave as full unification. In the first example, if we put the equation, `(x2 voice) = active`, before the equation, `(x0 = x2)`, then it works with no problem. For the second example, there are several ways to rewrite the rules so that pseudo unification rejects the sentence.

A.8.5 Summary

In this section, we described pseudo unification, and its advantages and disadvantages were discussed. It is certainly the case that pseudo-unification lacks the theoretical elegance of full-unification. Nevertheless, we feel that pseudo unification is still attractive for those whose primary concern is practical applications rather than theoretical elegance.


```
(N <==> (d o g s)        ; RULE 1
   (((x0 root) = dog)
    ((x0 agr number) = plural)
    ((x0 agr mind) = animate)))

(V <==> (r u n)          ; RULE 2
   (((x0 root) = run)
    ((x0 agr number) = plural)))

(S <==> (N V)            ; RULE 3
   ((x0 = x2)
    ((x0 subj) = x1)
    ((x0 agr) = (x1 agr))))

(S1 <==> (S)             ; RULE 4
   ((x0 = x1)
    ((x0 agr gender) = fem)
    ((x0 subj agr gender) = masc)))
```

Figure A.4: Counter Example II

After all, the choice is the user's. PSEUDO and FULL unification modes can be selected by setting the variable `*unification-mode*` to be `PSEUDO` or `FULL`, respectively. The default is `PSEUDO`.

Appendix B

GENKIT and TRANSKIT Version 3-2: User's Manual

GENKIT is a system that compiles a grammar into a sentence generation program [1]. The grammar is written in a formalism called Pseudo Unification Grammar. The compiled grammar is a standard lisp file consisting of a bunch of function definitions (DEFUN's).

TRANSKIT is a system that compiles a transformation rule into a lisp program that performs tree-to-tree (frame-to-frame and f-structure to f-structure) transformation. The rule of the transformation can be written in the same formalism as the GENKIT, i.e., Pseudo Unification Grammar.

GENKIT / TRANSKIT release version 3-2 is implemented in Common Lisp and no window graphics is used; thus the system is transportable, in principle, to any machines that support Common Lisp.

Those who are interested in obtaining the software described in this document should contact:

Radha Rao
Business Manager
Center for Machine Translation
Carnegie-Mellon University
Pittsburgh, PA15213, USA
rdr@nl.cs.cmu.edu

Several members of CMU Center for Machine Translation have made contributions to the system development. Phil Franklin and Ron Grider have developed earlier versions of this system. Eric Nyberg and Steve Morrisson have maintained the earlier versions of the system. The version 3-2 has been written by Masaru Tomita. Some part of this document is written by Ron Grider and Steve Morrisson.

Funding for this project is provided by several private institutions and governmental agencies in the United States and Japan.

[1] This appendix is based on a previously published technical report [■]. I would like to acknowledge the coauthor, Eric Nyberg, whose contributions are included in this appendix.

B.1 Getting Started

The *Generalized LR Parser/Compiler version 8-4* (or later) includes GENKIT and TRANSKIT. Thus, to use GENKIT, load the generalized LR Parser/compiler.

B.1.1 Basic Functions of GENKIT

(COMPGEN *file-name*)
COMPGEN takes a file name (e.g. "gra1") and compiles the file with extension ".gra" (e.g. "gra1.gra"), producing a file with the suffix "_gen.lisp" (e.g. "gra1_gen.lisp"). The "_gen.lisp" is a standard lisp program file; it can be simply loaded using the LOAD function. The "_gen.lisp" file can be compiled into machine code using the COMPILE-FILE function. The "_gen.lisp" file is automatically load each time you call COMPGEN function.

(GENERATOR *f-structure*)
After loading the "_gen" file, a sentence can be generated using the top level function called GENERATOR, that takes a f-structure of the sentence being generated.

(TR *string*)
This function is used to perform translation, interfacing to the Generalized LR Parser/Compiler. The sentence "*string*" is parsed and the parser's output (a f-structure) is fed to the function GENERATOR.

B.1.2 Basic Functions of TRANSKIT

(COMPTRF *file-name*)
COMPTRF takes a file name (e.g. "sem-map") and compiles the file with extension ".gra" (e.g. "sem-map.gra"), producing a file with the suffix "_trf.lisp". The "_trf.lisp" is a standard lisp program file; it can be simply loaded using the LOAD function. The "_trf.lisp" file can be compiled into machine code using the COMPILE-FILE function. The "_trf.lisp" file is automatically load each time you call COMPTRF function.


```
(<DEC> <==> (<NP> <VP>)
      (((x1 case) = nom)
       ((x2 form) =c finite)
       (*OR*
         (((x2 :time) = present)
          ((x1 agr) = (x2 agr)))
         (((x2 :time = past)))
       (x0 = x2)
       ((x0 subj) = x1)
       ((x0 passive) = -)))
```

Figure B.1: A Grammar Rule for Parsing

B.2 Writing a Generation Grammar

The grammar formalism for GENKIT is called *Pseudo Unification Grammar*, which is the same formalism as in the Generalized LR Parser/Compiler. The Pseudo Unification Grammar formalism resembles that of PATR-II. The following rule is an example Pseudo Unification Grammar rule for parsing (not for generation). Each rule consists of a context-free phrase structure description and a cluster of *pseudo equations*. The non-terminals in the phrase structure part of the rule are referenced in the constraint equations as x0...xn, where x0 is the non-terminal in the left hand side (here, <DEC>) and xn is the n-th non-terminal in the right hand side (here, x1 represents <NP> and x2 represents <VP>). The pseudo equations are used to check certain attribute values, such as verb form and person agreement, and to construct a f-structure.

In parsing, these rules are used to combine one or more constituents into a higher constituent. In generation, on the other hand, these rules are used to disassemble a constituent (left hand side) into several lower constituents (right hand side). We need a different set of pseudo equations for generation. It is possible and interesting to derive pseudo equations for generation automatically from those for parsing, but it is beyond the scope of this document. The following rule is the generation rule which corresponds to B.1. Descriptions of pseudo equations shall be presented later in this document. GENKIT takes this kind of grammar rules, and generates a LISP program that implements a run-time sentence generation.

```
(<DEC> <==> (<NP> <VP>)
      (((x0 passive) = -)
       (x1 == (x0 subj))
       (x2 = x0)
       (*OR*
         (((x2 :time) = present)
          ((x1 agr) = (x2 agr)))
         (((x2 :time = past)))
       ((x2 form) = finite)
       ((x1 case) = nom)))
```

Figure B.2: A Grammar Rule for Generation

The syntactic structures produced by the parser and the syntactic structures accepted as input by the generator are identical in form. These structures are called *f-structures*. Here is an example of the f-structure for the English sentence "I have a pain in my head": The f-structure captures the constituent structure of the utterance, the lexical entries (roots) of the constituents, and the tense and agreement features of the constituents where appropriate.

The process of generating a surface utterance (i.e., a string of words) from a syntactic f-structure is basically the reverse of parsing. In the sample grammar rule, a constraint equation places the information from the <NP> inside the subject of the <DEC> during parsing; in generation, the f-structure for a <DEC> will be broken up into its constituent parts, each of which will be generated by further recursive applications of grammar rules. In this case, the embedded f-structure that fills the subject slot of the declarative f-structure will be used as input to all of the rules that can possibly generate an <NP>. The generator follows a top-down, depth-first strategy for applying rules during generation. If the current search path fails, the generator backs up to the next applicable rule. This process continues until a successful generation is found, or until all of the rules are exhausted.

The current implementation of the generator compiler involves creating a set of LISP functions which represents the grammar of the target language. Each function, GG-X (where X is any syntactic category), implements all rewrite rules from the grammar whose left hand symbol is <X>. When GG-X is called with the f-structure representation of a source-language string, if that string can be generated by expansion of the non-terminal <X>, then


```
((root have)
 (time present)
 (subject ((root I)
           (agr 1sg)))
 (object ((root pain)
          (det ((root a)))
          (ppadjunct ((root head)
                      (poss ((root my)))
                      (prep ((root in)))
                      (agr 3sg)))
          (agr 3sg))))
```

Figure B.3: A Sample F-structure

GG-X returns the representative target-language string.

The process of constructing "GG-" functions consists of reading rewrite rules from the file containing the target-language grammar, and adjusting the appropriate "GG-" function after each rule is read. The first time that a rewrite rule for <X> is read, the basic shell of the GG-X function is created, and the LISP code implementing that rule is added, as a clause of OR (the argument passed to GG-X, x0, is to be an f-structure):

```
(defun  GG-X  (x0)
  (OR  **LISP code for first rewrite rule** ))
```

This new function is stored in a list with the other "GG-" functions. Each time a rewrite rule for <X> is read from the grammar file, function GG-X is retrieved from this list, and the code for the new rule is added as the last argument to the OR predicate. The finished version of GG-X is of the following form:

```
(defun  GG-X  (x0)
  (OR  **LISP code for the first rewrite rule**
       **LISP code for the second rewrite rule**
                          .
                          .
                          .
       **LISP code for the last rewrite rule** ))
```

Each rule is in the following form.

(*lhs-symbol arrow rhs-symbols list-of-pseudo-equations*)

The left hand side, *lhs-symbol*, must be a non-terminal symbol (e.g. `<S>`, `<NP>`). When there are more than one rule with the same left hand side symbol, More preferred rules should be present first, as rules will be applied from the top to bottom at runtime.

The *arrow* has to be either `==>` or `-->`. If `==>` is used, it will generate a space between the right hand side constituents. If `-->` is used, no space will be generated.

The right hand symbol in *rhs-symbols* must be one of the following three:

- Non-terminal symbol – the system will call its GG function recursively.
- Terminal symbol – the system will generate the terminal symbol.
- Wild card symbol (`%`) – the system will generate a string (or a symbol) in its `value` slot.

The fourth element, *list-of-pseudo-equations*, is described in section B.4.

B.3 Writing TRANSKIT rules

Each transformation rule is in the following form.
(*goalobj* `<==` *sourceobj list-of-pseudo-equations*)

The goal object name, *goalobj*, should be some appropriate name describing the object which the rule is supposed to produce.

The source object name, *source*, should be some appropriate name describing the object from which the rule is supposed to transform.

The rule body, *list-of-pseudo-equations*, is a list of pseudo equations written in exactly the same way as the Pseudo Unification Grammar in GENKIT. *goalobj* is represented as "x0", and *sourceobj* is represented as "x1". Pseudo equaitons are described in detail in section B.4.

The `COMPTRF` function compiles each rule in this form and produces a lisp program whose name is "*goalobj*`-from-`*sourceobj*". For example, the following transformation rule:

```
(deep-sem <== eng-sem
  ((x0 = x1)
   (*EOR*
        (((x1 cfname) = *utilize)
         (*OR*
                 (((x1 obj cfname) = *bath)
                  ((x0 cfname) <= '*bathe-action))
                 (((x1 obj cfname) = *shower)
                  ((x0 cfname) <= *shower-action)))
         ((x0 obj) = *REMOVE*))
        (((x1 cfname) = *.....
  )))
```

will be compiled into the following function definition:

```
 (defun deep-sem-from-eng-sem (eng-sem)
    ....{\it some serious lisp code).....)
```

and this function definition will be written in a "_trf.lisp" file. After loading the "_trf.lisp", the function can be used to transform one f-structure to another as in the following example:

```
(deep-sem-from-eng-sem '((cfname *utilize)
                         (obj ((cfname *bath)))
                         (agent ((cfname *person)
```

```
                                (name john))))     )

returns  ((cfname *bathe-action)
          (agent ((cfname *person)
                  (name john))))
```


B.4 Pseudo Equations

This section describes pseudo equations of the Pseudo Unification Grammar used in GENKIT and TRANSKIT, which is the same as the one used in the Generalized LR Parser/Compiler.

B.4.1 Basic Pseudo Equations

Pseudo Unification, =

path = *val*
Get a value from *path*, unify it with *val*, and assign the unified value back to *path*. If the unification fails, this equation fails. If the value of *path* is undefined, this equation behaves like a simple assignment. If *path* has a value, then this equation behaves like a test statement.

path1 = *path2*
Get values from *path1* and *path2*, unify them, and assign the unified value back to *path1* and *path2*. If the unification fails, this equation fails. If both *path1* and *path2* have a value, then this equation behaves like a test statement. If the value of *path1* is not defined, this equation behaves like a simple assignment.

Overwrite Assignment, <=

path <= *val*
Assign *val* to the slot *path*. If *path1* is already defined, the old value is simply overwritten.

path1 <= *path2*
Get a value from *path2*, and assign the value to *path1*. If *path1* is already defined, the old value is simply overwritten.

path <= *lisp-function-call*
Evaluate *lisp-function-call*, and assign the returned value to *path*. If *path1* is already defined, the old value is simply overwritten. *lisp-function-call* can be an arbitrary lisp code, as long as all functions called in *lisp-function-call* are defined. A path can be used as a special function that returns a value of the slot.

Removal Assignment, ==

path1 == *path2*
Get a value from *path2*, assign the value to *path1*, and remove the value of *path2* (assign nil to *path2*). If a value already exists in *path1*, then the new value is unified with the old value. If the unification fails, then this equation fails.

No lisp function can be written in the right hand side.

Append Multiple Value, >

path1 > *path2*
Get a value from *path2*, and assign the value to *path1*. If a value already exists in *path1*, the new value is appended to the old value. The resulting value of *path1* is a multiple value.

Pop Multiple Value, <

path1 < *path2*
The value of *path2* should be a multiple value. The first element of the multiple value is popped off, and assign the value to *path1*. If *path1* already has a value, unify the new value with the old value. If *path2* is undefined, this equation fails.

DEFINED and *UNDEFINED*

path = *DEFINED*
Check if the value of *path* is defined. If undefined, then this equation fails. If defined, do nothing.

Constraint Equations, =c

path =c *val*
This equation is the same as Pseudo Unification
path = *val* except if *path* is not already defined, it fails. Thus, the constraint equation is roughly the same as the following sequence of equations.

path = *DEFINED*
path = *val*

Removing Values, *REMOVE*

path = *REMOVE*
This equation removes the value in *path*, and the path becomes undefined.

B.4.2 Special Forms

Disjunctive Equations, *OR*

(*OR* *list-of-equations list-of-equations* ....)
All lists of equations are evaluated disjunctively. This is an inclusive OR, as oppose to exclusive OR; Even if one of the lists of equations is evaluated successfully, the rest of lists will be also evaluated anyway.

Exclusive OR, *EOR*

(*EOR* *list-of-equations list-of-equations* ....)
This is the same as disjunctive equations *OR*, except an exclusive OR is used. That is, as soon as one of the element is evaluated successfully, the rest of elements will be ignored.

Case Statement, *CASE*

(*CASE* *path* (*key1 equation1-1 equation1-2* ...) (*Key2 equation2-1* ...) (*Key3 equation3-1*) ....)
The *CASE* statement first gets the value in *path*. The value is then compared with Key1, Key2, ..., and as soon as the value is eq to some key, its rest of equations are evaluated. This is similar to Common Lisp case form, except Keys cannot be a list of objects. Example:

```
(*CASE* (x1 root)
  (have ......)
  (be ......)
  (take ......))
```

(*CASE* *lisp-function-call* (*key1 equation1-1 equation1-2* ...) (*Key2 equation2-1* ...) (*Key3 equation3-1*) ....)
One can write an arbitrary lisp function call in stead of a path.

Test with Arbitrary LISP Function, *TEST*

(*TEST* *lisp-function-call*)
The *lisp-function-call* is evaluated, and if the function returns nil, it fails. If the function returns a non-nil value, do nothing. A path can be used as special function that returns a value of the slot. Thus,
(*TEST* *path*)
has the same effects as
path = *DEFINED*
and
(*TEST* (NOT *path*))
has the same effects as
path = *UNDEFINED*

Recursive Evaluation of Equations, *INTERPRET*

(*INTERPRET* *path*)
The *INTERPRET* statement first gets a value from *path*. The value of *path* must be a valid list of equations. Those equations are then recursively evaluated. This *INTERPRET* statement resembles the "eval" function in Lisp.

(*INTERPRET* *lisp-function-call*)
First, *lisp-function-call* is evaluated. The *lisp-function-call* must return a valid list of equations. Those equations are then evaluated recursively.

B.4.3 Special Values

Disjunctive Value, *OR*

(*OR* *val val ...*)
Unification of two disjunctive values is set interaction. For example, (unify '(*OR* a b c d) '(*OR* b d e f)) is (*OR* b d).

Negative Value, *NOT*

(*NOT* *val val ...*)
Unification of two negative values is set union. For example, (unify '(*NOT* a b c d) '(*NOT* b d e f)) is (*NOT* a b c d e f).

Unification of a disjunctive value and a negative value is set substruction. For example, (unify '(*OR* a b c d) '(*NOT* b d e f)) is (*OR* a c).

Multiple Values, *MULTIPLE*

(*MULTIPLE* *val val ...*)
Unification of two multiple values is append.

When unified with a value, each element is unified with a value. For example, (unify '(*MULTIPLE* a b c d b d e f) 'd) is (*MULTIPLE* d d).

When unified with a disjunctive value, the result is a disjunction of multiple values. For example, (unify '(*MULTIPLE* a b c d b d e f) '(*OR* b d e f g))) (*OR* (*MULTIPLE* b b) (*MULTIPLE* d d) e f).

User Defined special Values, **user-defined**

The user can define his own special values. An unification function with the name UNIFY**user-defined** must be defined. The function should take two arguments, and returns the new value or *FAIL* if the unification fails. For example, to define *ISA* value,

```
(defun unify*isa* (x y)
  (cond ((isa-p x y) x)
        ((isa-p y x) y)
        (t '*FAIL*)))
```

where isa-p is some function to look up the is-a relation of two objects in a is-a hierarchy.

B.5 Compiling a Grammar in Multiple Files

The user can compile sub files with COMPGEN and load them with LOAD separately, as long as the following conditions are satisfied.

- A set of rules with the same left hand side must be in a single file. No two rules in two different files must have the same left hand side.
- The system assumes that the last-compiled file is the main grammar file.

B.6 Sample GENKIT Grammar

```
;;;
;;; This a test generation grammar by Steve Morrisson.
;;;

(<start> ==> (<dec> <period>)
         (((x1 mood) = imperative)
          (x1 = x0)))

(<period> --> (p e r i o d)
          ())

(<dec> ==> (<np> <vp>)
       ((x1 == (x0 subj))
        (x2 = x0)))

;;; VP
(<vp> ==> (<vp> <pp>)
      ((x2 < (x0 ppadjunct))
       (x1 = x0)))

(<vp> ==> (<vp> <advP>)
      ((x2 < (x0 advadjunct))
       (x1 = x0)))

(<vp> ==> (<v> <np>)
      ((x2 == (x0 object))
       (x1 = x0)))

(<vp> ==> (<v>)
      ((x1 = x0)))

(<v> ==> (%)
     (((x1 value) <= (morph-root-verb (x0)))))

;;; NP
(<np> ==> (<prep> <np)
      ((x2 < (x0 ppadjunct))
```



```
       (x1 = x0)))

(<np> ==> (<n1>)
      ((x1 = x0)))

(<n1> ==> (<det> <n1>)
      ((x1 == (x0 det))
       (x2 = x0)))

(<n1> ==> (<adjP> <n1>)
      ((x1 < (x0 adjadjunct))
       (x2 = x0)))

(<n1> ==> (<n1> <pp>)
      ((x2 < (x0 ppadjunct))
       (x1 = x0)))

(<n1> ==> (<n>)
      ((x1 = x0)))

(<n> ==> (%)
     (((x1 value) <= (morph-root-noun (x0 root) (x0 count)))))

(<pp> ==> (<prep> <np>)
      ((x1 == (x0 prep))
       (x2 = (x0 obj))))

(<prep> ==> (%)
        (((x1 value) = (x0 root))))

(<det> ==> (%)
       (((x1 value) = (x0 root))))
```

Bibliography

[Aho and Ullman, 1972] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II. Prentice-Hall, Englewood Cliffs, N. J., 1972.

[Aho and Ullman, 1977] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977.

[Bates and Lavie, 1991] J. Bates and A. Lavie. Recognizing Substrings of LR(k) Languages in Linear Time. *Technical Report CMU-CS-91-188*, 1991.

[Bates and Lavie, 1992] J. Bates and A. Lavie. Recognizing Substrings of LR(k) Languages in Linear Time. In *Proceedings of POPL'92*, Albuquerque, NM, 1992. ACM press.

[Bresnan and Kaplan, 1982] J. Bresnan and R. Kaplan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages pp. 173–281. MIT Press, Cambridge, Massachusetts, 1982.

[Buø, in preparation] F. D. Buø. A learnable connectionist parser that outputs f-structures (working title). Master's thesis, University of Karlsruhe, in preparation.

[Carbonell and Hayes, 1984] J. G. Carbonell and P. J. Hayes. Recovery strategies for parsing extragrammatical language. Technical Report CMU-CS-84-107, Computer Science Department, Carnegie-Mellon University, Feb 1984.

[Carbonell and Tomita, 1985] Jaime G. Carbonell and Masaru Tomita. New approaches to machine translation. Technical report, School of Computer Science, Carnegie Mellon University, 1985.

[Chow and Roukos, 1989] Y. Chow and S. Roukos. Speech Understanding Using a Unification Grammar. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'89)*, pages 727–730, 1989.

[Chow *et al.*, 1987] Y. Chow, M. Dunham, O. Kimball, M. Krasner, G. Kubala, J. Makhoul, S. Roucos, and R. Schwartz. BYBLOS: The BBN continuous speech recognition system. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-87*)*, pages 89–92, Dallas, 1987.

[Chvátal, 1983] Vašek Chvátal. *Linear Programming*, chapter 6. 1983.

[Cole *et al.*, 1983] R. Cole, R. Stern, M. Phillips, S. Brill, P. Specker, and A. Pilant. Feature-based speaker independent recognition of english letters. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-83*)*, pages 731–783, 1983.

[Earley, 1970] J. Earley. An efficient context-free parsing algorithm. *Communication of ACM*, 6(8):94–102, February 1970.

[Fain *et al.*, 1985] J. Fain, J.G. Carbonell, P. Hayes, and S. Minton. MULTIPAR: A robust entity-oriented parser. In *Proceedings of the Seventh Cognitive Science Society Conference*, pages 110–119, Irvine, Calif., 1985.

[Fn and Booth, 1975] K. S. Fu and T. L. Booth. Grammatical inference: Introduction and survey — part ii. *IEEE Trans on Sys., Man and Cyber*, SMC-5:409–423, 1975.

[Fujisaki, 1984] T. Fujisaki. An approach to stochastic parsing. In *Proceedings of COLING84*, 1984.

[Hanazawa *et al.*, 1990] T. Hanazawa, K. Kita, S. Nakamura, T. Kawabata, and K. Shikano. ATR HMM-LR continuous speech recognition system. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-90*)*, Albuquerque, N. Mex., 1990.

[Itakura, 1975] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 23(1):67–72, 1975.

[Jain, 1991] A.J. Jain. Parsing complex sentences with structured connectionist networks. *Neural Computation*, 3:110–120, 1991.

[Johnson, 1975] S. C. Johnson. Yacc – yet another compiler compiler. Technical Report CSTR 32, Bell Laboratories, 1975.

[Karttunen, 1986] L. Karttunen. D-patr: A development environment for unification-based grammars. In *12th International Conference on Computational Linguistics*. Bonn, 1986.

[Kasami, 1965] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[Kay, 1984] M. Kay. Functional unification grammar: A formalism for machine translation. In *10th International Conference on Computational Linguistics*, pages 75–78, Stanford, July 1984.

[Kiparsky, 1985] C. Kiparsky. Lfg manual. Technical report, Xerox Palo Alto Research Center, 1985.

[Kita *et al.*, 1989a] K. Kita, T. Kawabata, and H. Saito. HMM Continuous Speech Recognition Using Predictive LR Parsing. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1989.

[Kita *et al.*, 1989b] K. Kita, T. Kawabata, and H. Saito. HMM continuous speech recognition using predictive lr parsing. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-89*)*, pages 53–56, Glasgow, N. Mex., 1989.

[Kitano *et al.*, 1989] Hiroaki Kitano, Teruko Mitamura, and Masaru Tomita. Massively parallel parsing in PhiDM Dialog: Integrated architecture for parsing speech inputs. In *1st International Workshop on Parsing Technologies*, 1989.

[Kubala *et al.*, 1988] G. Kubala, Y. Chow, M. Derr, M. Feng, O. Kimball, J. Makhoul, P. Price, J. Rohlicek, S. Roucos, R. Schwartz, and J. Vandegrift. Continuous speech recognition results of the BYBLOS system on the DARPA 1000-word resource management database. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-88*)*, pages 291–294, New York, 1988.

[Laboratories, 1989] ATR Interpreting Telephony Research Laboratories. Research activities of the natural language nnderstanding department and

the knowledge and data base department for 1988. Technical Report TR-I-0070, ATR Interpreting Telephony Research Laboratories, Tokyo, 1989.

[Lang, 1974] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In G. Goos and J. Hartmanis, editors, *Proceedings of 2nd Colloquium on Automata, Languages and Programming*. Springer-Verlag Berlin, 1974. Lecture Notes in Computer Science (14).

[Lavie and Tomita, 1993a] Alon Lavie and Masaru Tomita. Efficient generalized LR parsing of word lattices. In *Third Bar-Ilan Symposium on the Foundation of Artificial Intelligence (BISFAI-93)*, Israel, 1993.

[Lavie and Tomita, 1993b] Alon Lavie and Masaru Tomita. Efficient generalized LR parsing of word lattices. *Journal of Mathematics and Artificial Intelligence*, 1993. to appear.

[Lavie and Tomita, 1993c] Alon Lavie and Masaru Tomita. GLR* - an efficient noise-skipping parsing algorithm for context-free grammars. In *Third International Workshop on Parsing Technologies (IWPT93)*, Tilburg and Belgium, 1993.

[Lea, 1980] W. Lea, editor. *Trends in Speech Recognition*. Prentice-Hall, 1980.

[Lee, 1988] K.-F. Lee. *Large-Vocabulary Speaker-Independent Continuous Speech Recognition: The Sphinx System*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[Lesser *et al.*, 1975] V. Lesser, R. Fennell, L. Erman, and R. Reddy. The hearsay-ii speech understanding system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 23(1), 1975.

[Lowerre and Reddy, 1980] B. Lowerre and R. Reddy. The HARPY speech recognition system. In W. Lea, editor, *Trends in Speech Recognition*, pages 340–360. Prentice-Hall, 1980.

[Lowerre, 1976] B. Lowerre. *The HARPY Speech Recognition System*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1976.

[Morii *et al.*, 1985] S. Morii, K. Niyada, S. Fujii, and M. Hoshimi. Large vocabulary speaker-independent japanese speech recognition system. In

*Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-85*)*, pages 866–869, Tampa, Fla., 1985.

[Ng and Tomita, 1991] See-Kiong Ng and Masaru Tomita. Probabilistic LR parsing for general context-free grammars. In *2nd International Workshop on Parsing Technologies (IWPT91)*, Cancun, Mexico, 1991.

[Nirenburg *et al.*, 1991] Sergei Nirenburg, Jaime G. Carbonell, Masaru Tomita, and Ken Goodman. *Knowledge-Based Machine Translation*. Morgan Kaufmann Publishers, 1991.

[Osterholtz *et al.*, 1992] L. Osterholtz, A. McNair, I. Rogina, H. Saito, T. Sloboda, J. Tebelskis, A. Waibel, and M. Woszczyna. Testing generality in JANUS: a multi-lingual speech to speech translation system. In *ICASSP92*, volume 1, pages 209–212, 1992.

[Pereira, 1985] F. C. N. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Chicago, July 1985.

[Polzin, in preparation] T.S. Polzin. Pronoun resolution. interaction of syntactic andsemantic information in connectionist parsing. Master's thesis, Carnegie Mellon University, Department of Philosophy, Computational Linguistics, in preparation.

[Rabiner *et al.*, 1988] L. Rabiner, J. Wilpon, and F. Soong. High performance connected digit recognition using hidden markov models. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-88*)*, pages 1214–1225, New York, 1988.

[Rekers and Koorn, 1991] J. Rekers and W. Koorn. Substring Parsing for Arbitrary Context-Free Grammars. In *Proceedings of Second International Workshop on Parsing Technologies*, pages 218–224, Cancun, Mexico, 1991.

[Saito and Tomita, 1988a] H. Saito and M. Tomita. Parsing Noisy Sentences. In *Proceedings of 12th International Conference on Computational Linguistics (COLING)*, Budapest, Hungary, 1988.

[Saito and Tomita, 1988b] Hiroaki Saito and Masaru Tomita. Parsing noisy sentences. In *12th International Conference on Computational Linguistics (COLING88)*, Budapest, 1988.

[Saito, 1990] H. Saito. Bi-directional LR Parsing from an Anchor Word for Speech Recognition. In *Proceedings of 13th International Conference on Computational Linguistics (COLING)*, Helsinki, Finland, 1990.

[Seneff, 1992] S. Seneff. A relaxation method for understanding spontaneous speech utterances. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 299–304, February 1992.

[Shieber, 1984] S. M. Shieber. The design of a computer language for linguistic information. In *10th International Conference on Computational Linguistics*, pages 362–366, Stanford, July 1984.

[Shieber, 1985] S. M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago, July 1985.

[Shieber, 1986] S. M. Shieber. *An Introduction to Unification Approaches to Grammar*. CSLI Lecture Notes. Center for the Study of Language and Information, 1986.

[Stallard and Bobrow, 1992] D. Stallard and R. Bobrow. Fragment processing in the DELPHI system. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 305–310, February 1992.

[Stentiford and Steer, 1988] F. Stentiford and M. Steer. Machine translation of speech. *British Telecom Technology Journal*, 6(2), 1988.

[Strang, 1980] G. Strang. *Linear Algebra and Its Applications*. Academic Press, New York, NY, 2nd edition, 1980.

[Suppes, 1970] P. Suppes. Probabilistic grammars for natural languages. *Synthese*, 22:95–116, 1970.

[Tebelskis and Waibel, 1990] J. Tebelskis and A. Waibel. Large vocabulary recognition using linked predictive neural networks. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (*ICASSP-90*)*, pages 437–440, Albuquerque, N. Mex., 1990.

[Thompson, 1989] H. Thompson. A Chart Parsing Realisation of Dynamic Programming, with Best-first Enumeration of Paths in a Lattice. In *Proceedings of European Conference on Speech Communication and Technology (Eurospeech'89)*, pages 378–381, Paris, France, September 1989.

[Thompson, 1990] H. Thompson. Best-first Enumeration of Paths through a Lattice - an Active Chart Parsing Solution. *Computer Speech and Language*, 4(3):263 274, 1990.

[Tomabechi and Tomita, 1989] Hideto Tomabechi and Masaru Tomita. The direct memory access paradigm and its application to natural language processing. *Computers and Artificial Intelligence*, 8(5), 1989.

[Tomabechi *et al.*, 1989] Hideto Tomabechi, Hiroaki Saito, and Masaru Tomita. SpeechTrans: An experimental real-time speech-to-speech translation system. In *AAAI Spring Symposium on Spoken Language Systems*, 1989.

[Tomita and Carbonell, 1986] Masaru Tomita and Jaime G. Carbonell. Another stride toward knowledge based machine translation. In *11th International Conference on Computational Linguistics (COLING86)*, Bonn, 1986.

[Tomita and Carbonell, 1987a] Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. Technical Report CMU-CMT-87-101, Center for Machine Translation, Carnegie Mellon University, 1987.

[Tomita and Carbonell, 1987b] Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *10th International Joint Conference on Artificial Intelligence (IJCAI87)*, Milano, Italy, 1987.

[Tomita and Carbonell, 1988] Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *1986/1987 Research Review*. School of Computer Science, Carnegie Mellon University, 1988.

[Tomita and Knight, 1988] M. Tomita and K. Knight. Pseudo unification and full unification. Technical Report unpublished, Center for Machine Translation, Carnegie Mellon University, 1988.

[Tomita and Ng, 1991] Masaru Tomita and See-Kiong Ng. The generalized LR parsing algorithm. In Masaru Tomita, editor, *Generalized LR Parsing*. Kluwer Academic Publishers, Boston MA, 1991.

[Tomita and Nyberg, 1988] Masaru Tomita and Eric Nyberg. The Generation Kit and the Transformation Kit: User's guide. Technical report, Center for Machine Translation, Carnegie Mellon University, 1988.

[Tomita *et al.*, 1987] Masaru Tomita, Marion Kee, Teruko Mitamura, and Jaime G. Carbonell. Linguistic and domain knowledge sources for the universal parser architecture. In *International Congress on Terminology and Knowledge Engineering*, Trier, 1987.

[Tomita *et al.*, 1988a] Masaru Tomita, Marion Kee, Hiroaki Saito, Teruko Mitamura, and Hideto Tomabechi. The universal parser compiler and its application to a speech translation system. In *2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Pittsburgh, Pennsylvania, 1988.

[Tomita *et al.*, 1988b] Masaru Tomita, Teruko Mitamura, Hiroyuki Musha, and Marion Kee. The generalized LR parser/compiler version 8.1: User's guide. Technical report, Center for Machine Translation, Carnegie Mellon University, 1988.

[Tomita *et al.*, 1989] Masaru Tomita, Marion Kee, Hiroaki Saito, Teruko Mitamura, and Hideto Tomabechi. Towards a speech-to-speech translation system. *Journal of Applied Linguistics*, 3(1), 1989.

[Tomita *et al.*, 1990a] Masaru Tomita, Hideto Tomabechi, and Hiroaki Saito. Speech trans: An experimental real-time speech-to-speech translation system. *Language Research*, 26(4), 1990.

[Tomita *et al.*, 1990b] Masaru Tomita, Hideto Tomabechi, and Hiroaki Saito. SpeechTrans: An experimental real-time speech-to-speech translation system. In *International Conference on Natural Language Processing*, Seoul, 1990.

[Tomita, 1985] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston MA, 1985.

[Tomita, 1986] Masaru Tomita. An efficient word lattice parsing algorithm for continuous speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP86)*, Tokyo, 1986.

[Tomita, 1987] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2), 1987.

[Tomita, 1988a] Masaru Tomita. Combining lexicon-driven parsing and phrase-structure-based parsing. In *12th International Conference on Computational Linguistics (COLING88)*, Budapest, 1988.

[Tomita, 1988b] Masaru Tomita. Graph-structured stack and natural language parsing. In *26th Annual Meeting of the Association for Computational Linguistics (ACL88)*, Buffalo, New York, 1988.

[Tomita, 1988c] Masaru Tomita. Linguistic sentences and real sentences. In *12th International Conference on Computational Linguistics (COLING88)*, Budapest, 1988.

[Tomita, 1988d] Masaru Tomita. Parser factory: The universal parser compiler and its application to a knowledge-based speech translation system. In *13th International IAUD Symposium on Linguistic Approaches to Artificial Intelligence*, Duisburg, 1988.

[Tomita, 1988e] Masaru Tomita. Towards speech translation. In *2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Pittsburgh, Pennsylvania, 1988.

[Tomita, 1990a] Masaru Tomita, editor. *Current Issues in Parsing Technologies*. Kluwer Academic Publishers, Boston MA, 1990.

[Tomita, 1990b] Masaru Tomita. The generalized LR parser/compiler. In *13th International Conference on Computational Linguistics (COLING90)*, Helsinki, 1990.

[Tomita, 1991] Masaru Tomita, editor. *Generalized LR Parsing*. Kluwer Academic Publishers, Boston MA, 1991.

[Tomita, 1992] Masaru Tomita. Application of the TOEFL test to the evaluation of Japanese-English MT. In *NSF Workshop on MT Evaluation*, San Diego, 1992.

[Van der Steen, 1987] G.J. Van der Steen. A program generator for recognition, parsing and transduction with syntactic patterns. Technical report, vakgroep Alfa-informatica, Faculteit der Letteren, Universiteit van Amsterdam, 1987.

[Waibel *et al.*, 1991] A. Waibel, A. Jain, A. McNair, A. H. Saito, Hauptmann, and J. Tebelskis. JANUS: a speech-to-speech translation system using connectionist and symbolic processing strategies. In *ICASSP91*, volume 2, pages 793–796, 1991.

[Ward, 1989] W. Ward. Understanding spontaneous speech. In *DARPA Speech and Natural Language Workshop*, pages 137–141, 1989.

[Ward, 1990] W. Ward. The CMU air travel information service: Understanding spontaneous speech. In *DARPA Speech and Natural Language Workshop*, 1990.

[Ward, 1991] W. Ward. Understanding spontaneous speech: The Phoenix system. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 365–367, April 1991.

[Wetherall, 1980] C. S. Wetherall. Probabilistic languages: A review and some open questions. *Computing Surveys*, 12:361–379, 1980.

[Wolf and Woods, 1980] J. Wolf and W. Woods. The HWIM speech understanding system. In W. Lea, editor, *Trends in Speech Recognition*, pages 316–339. Prentice-Hall, 1980.

[Woods *et al.*, 1976] W. Woods, M. Bates, G. Brown, B. Bruce, C. Cook, J. Klovstad, J. Makhoul, B. Nash-Webber, R. Schwartz, J. Wolf, and V. Zue. Speech understanding systems—final technical report. Technical report, Bolt, Beranek, and Newman, Cambridge, Mass., 1976.

[Woszczyna *et al.*, 1993] M. Woszczyna, O.Barkai, N.Coccaro, A.Eisele, A.McNair, I.Rogina, C.P.Rose, T.Sloboda, M.Tomita, N.Aoki-Waibel, A.Waibel, and W. Ward. Recent advances in JANUS: CMU's speech translation system. In *ARPA Workshop on Human Language Technology*, Princeton, 1993.

[Wright and Wrigley, 1989] J. H. Wright and E. N. Wrigley. Probabilistic LR parsing for speech recognition. In *Proceedings of International Workshop on Parsing Technologies '89*, pages 105–114, 1989.

[Young *et al.*, 1989] S.R. Young, A.G. Hauptmann, W.H. Ward, E.T. Smith, and P. Werner. High level knowledge sources in usable speech recognition systems. *Communications of the ACM*, 32(2):183–194, 1989.